

Title: 3D handbook

Background

From an early passion for biology and science in high school to an in-depth study of genetics at university, my academic journey has been driven by a curiosity and commitment to understanding the natural world. After graduating, I ventured into various fields, including administration, property, and eventually coding. Each step, a testament to my adaptability and eagerness to learn, has equipped me with a diverse skill set. My career path showcases a blend of practical experience and intellectual exploration, demonstrating my versatility and dedication.

I am an independent thinker who thrives on self-directed learning and extensive research. Whether diving into complex scientific topics or exploring new technological advancements, I value the process of discovery and the pursuit of knowledge. My unwavering commitment to self-directed learning ensures that I stay updated with the latest developments in my field. I take pride in sharing insights and information with others, confident in my ability to adapt to changes.

Loyalty to a cause and a genuine passion for studying and teaching are central to my professional and personal life. By continuously seeking out new information and embracing diverse challenges, I strive to contribute meaningfully to every field I engage with. My love for learning is fueled by a strong desire to make an impact, inspiring me to push boundaries and create positive change.

Introduction.....	4
Geometry and Materials.....	7
Shaders.....	8
GLSL API.....	14
Types.....	14
Scalars.....	14
Vector Types.....	15
Matrix types.....	17
Textures, Mapping and Sampler Types.....	17
Types.....	20
Mathematical Functions.....	22
Geometric Functions.....	24
GLTF.....	28
Useful Concepts.....	29
Raycaster.....	29
Hulls.....	29
Concave & Convex Hulls.....	29
Barycentric Coordinates.....	30
dFdx && dFdy - GLSL partial function.....	31
RayMarch Algorithm.....	32
Compute shaders.....	33
Array Uniforms.....	33
Generating UV's Manually.....	33
Texture Feedback.....	34
OnBeforeCompile & Existing Materials.....	36
Case Studies.....	38
Depth textures, world coordinates and postprocessing.....	38
Mosaic Tiling.....	48
Subdivision modifier.....	53
Animated shader - alpha clipped grass.....	60
Ray marching over a Three.js scene.....	64
Volumetric lights and radial blur.....	70
GLSL and GPU Particles.....	78
Edge detection and concave hulls.....	80
[***] Images and positional data.....	85

[***] Procedural Flow maps.....	85
Procedural Low-Poly Clouds.....	85
Real time vertex displacement.....	85
GLSL Floodfill and dynamic shaders.....	85
Pulsating volume.....	86
**River shader and tube geometries.....	86
Spatial Clustering of Colors in Star Maps and Cell Images.....	86
Procedural Flow Maps.....	86
Appendix.....	87
1.1.....	88
1.2.....	91
1.3.....	93
1.4.....	98
1.5.....	100
1.5/1.6.....	102

Introduction

This book is a thrilling adventure for those who are eager to learn about shaders, 3D, and the render pipeline in WebGL. It delves into advanced 3D graphics concepts, guiding

you through the use of GLSL by tackling problems and finding creative solutions. The journey, set against the backdrop of Three.js, @react-three/fiber, WebGL, and shaders using GLSL, promises to be both challenging and rewarding.

Creativity is at the heart of any 3D project. It forces one to think outside the box in most situations with creative coding. In doing so, we will look at some exciting challenges and thoroughly work through solutions. So, not only will this book be a guide to use as a reference for situations requiring thinking about a specific topic, but it will also help with learning 3D and graphics from the ground up.

Why bother learning 3D when we have AI now, which can give pretty good implementations?

AI is here to stay and will help write about concepts. In the early days of ChatGPT (and other platforms), you had an AI that would rarely, if at all, give the correct answers. So, it was still necessary to learn 3D. Having said this, it is still essential to understand the concepts before you can become creative and wield AI effectively. Over the last few years, the age of implementation has been slowly coming to an end, and the age of ideas accompanying creativity has begun.

I hear you. That is a bold statement. So what do I mean? The endless hours of bug fixing and analyzing code for improvements will end, and even coming up with initial implementations will end. Eventually, when complete solutions are generated in seconds, the idea of working hard and endless hours into the night will end. The replacement: tweaking and bug-fixing AI code. So, where does creativity come in?

Twofold:

1. Using AI to solve problems creatively in incremental steps.
2. The most important aspect is to be an independent thinker.

On that second point of being an independent thinker, imagine after 50 years of AI regurgitating the exact old solutions it fed on through scraping the internet. We will need fresh and new ideas. Here is my point made concisely: Creatives and independent thinkers are needed more now than ever. When people stop creating new things and rely on AI, our civilization is slowly approaching death.

When faced with the question of why we should learn when we have AI prompts, the answer is clear. We need to learn to learn. This enables the creativity that comes with competence and expertise. The point of learning is not just what you learn, but how you learn to absorb information and express it effectively. Once you master this, you reach a sweet spot where you can confidently approach complicated problems as an independent thinker, fostering creativity.

There then becomes a sweet spot, and you learn how to learn, having the ability to solve complicated problems. You reinforce the idea of being an independent thinker, which is where creativity happens. When you have the confidence and knowledge to approach issues that haven't been solved before or very rarely, creativity happens. Having all the knowledge at one's fingertips allows one to think of new ways of doing things. This is even more so in tech. If you sit around for 5 minutes in tech, you will miss the next big thing. People who work in tech can often complain about the excess of choice. This shouldn't be frowned on; it should be celebrated and a precursor to what's coming in the next 2-6 years in every industry.

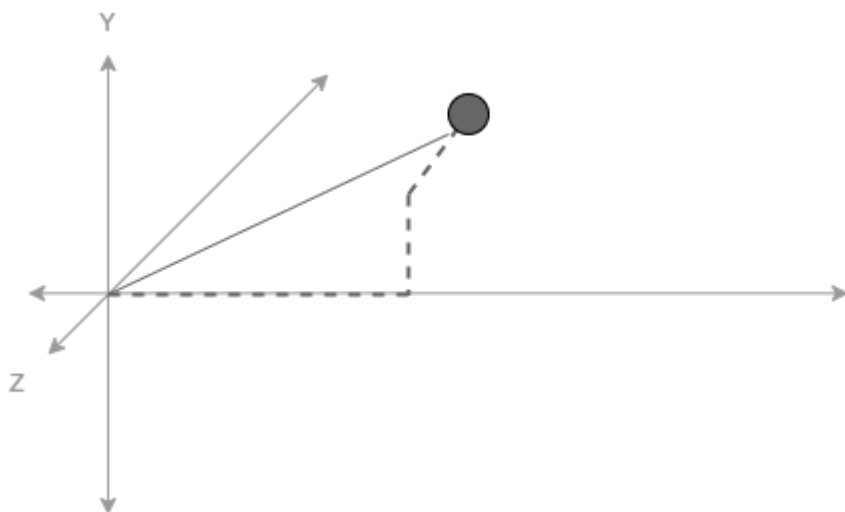
People need to know that it is not just about knowledge anymore but about creativity and new ideas. As stated above, implementations will soon be a thing of the past and algorithms and AI will develop entirely new coding methods. AI and computing power will explode in the coming 1 to 2 decades, advancing society in a way we have never seen before.

The age of the last 120 years is ending, and a new age of AI and creativity will flourish. It will not be enough to just think and regurgitate what textbooks say, as you will have that at your fingertips. Your job will be to think outside the box, come up with solutions people have not thought of before, or tweak solutions creatively. More of the same just will not be enough to survive in the coming age.

Geometry and Materials

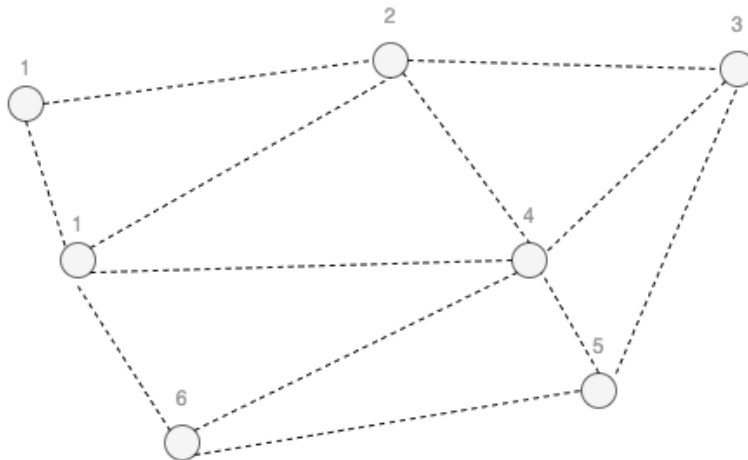
Geometries are ways of expressing shape in a computer generated math world.

Vectors are what define geometries, so what's a vector?



A vector is a map to a point in 3d space, it defines channels in what GLSL calls `vec2`'s, `vec3`'s and `vec4`'s. And goes from the origin zero to a point in 3D space.

Geometries are made of these vectors or vertices. Vertices are 3 component math entities which do exactly what's described above, direct to specific points in 3D space. Faces are then constructed around these vertices:



And in THREE.js there is a class called BufferGeometry which is optimized for vertices. It stores vertex data, face data via indices, UVs, normals and other vertex attributes.

So in the example above one face would require 3 indices to construct it. Therefore [1, 2, 4] would be one and [1, 4, 7] would be another. These indices are stored in a typed array. Javascript typed arrays are used for large amounts of data and store much more efficiently than non typed arrays. Its just a more optimized storage vessel for this data. But you could store any other numerical data in these typed arrays.

The indices typed array dictates how the faces form in the render pipeline. So it's important about the order. Every 3 indices creates a face. If they are random you will get random connections of faces.

Shaders

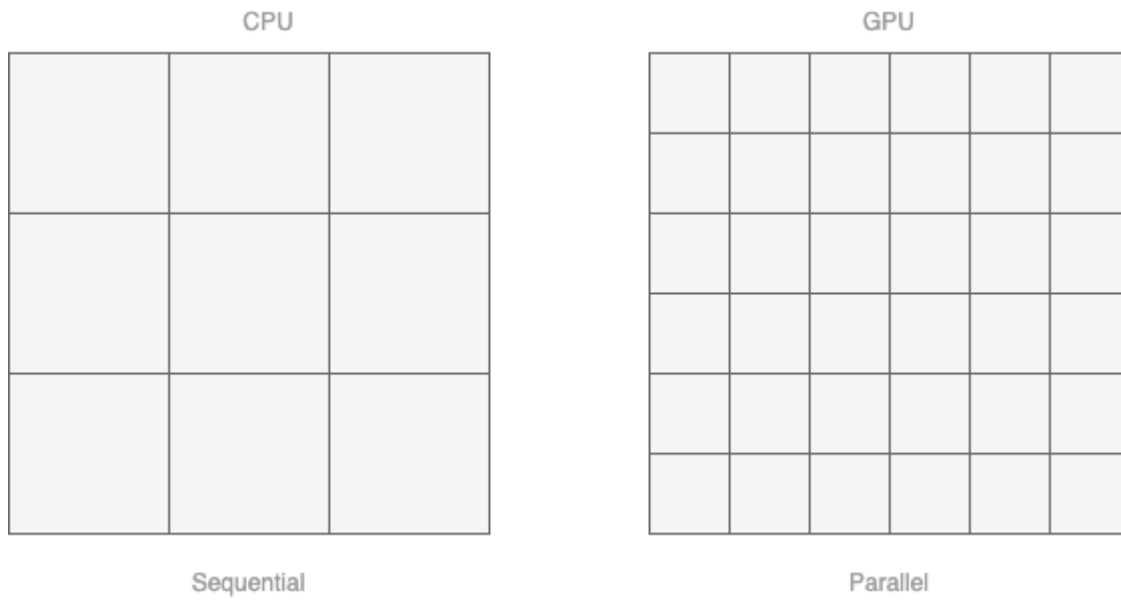
I mentioned the render pipeline above and this consists of several predetermined steps to go from a Mathematical 3D world to a mathematical 2D screen representation of this world. Here's the thing to remember, computer screens are not 3D..

They consist of a few hundred or thousand pixels and only have 2 dimensions, width and height, X and Y axes. So this begs the question - how do you go from 3 dimensional data to 2 dimensional data. Well first of all let's understand what shaders are, the order of what shaders run and then delve into coordinate systems - which facilitate this transition from 3D data to 2D data.

To talk about shaders we need to understand the CPU and GPU architecture. They fundamentally differ in their approach to processing data.

CPUs are sequential and GPUs are highly parallel environments. So the key difference in my experience is that CPUs allow for state. State in the sense of a centralized store where we can retrieve and utilize data.

Shaders and the GPU have no state and you have to use math and modulators. Like time or some other value passed to the shaders via attributes or uniforms. I'll go into more detail later on on this topic.



So imagine we have this:

```
// Vertex shader
attribute float specialValue;

void main () {
    float value = 0;

    value += specialValue;

    gl_Position = vec4(value, value, value, value);
}
```

This code runs in every red square at the same time (In the example above). But how does any pixel on the screen differ if the same inputs and outputs?

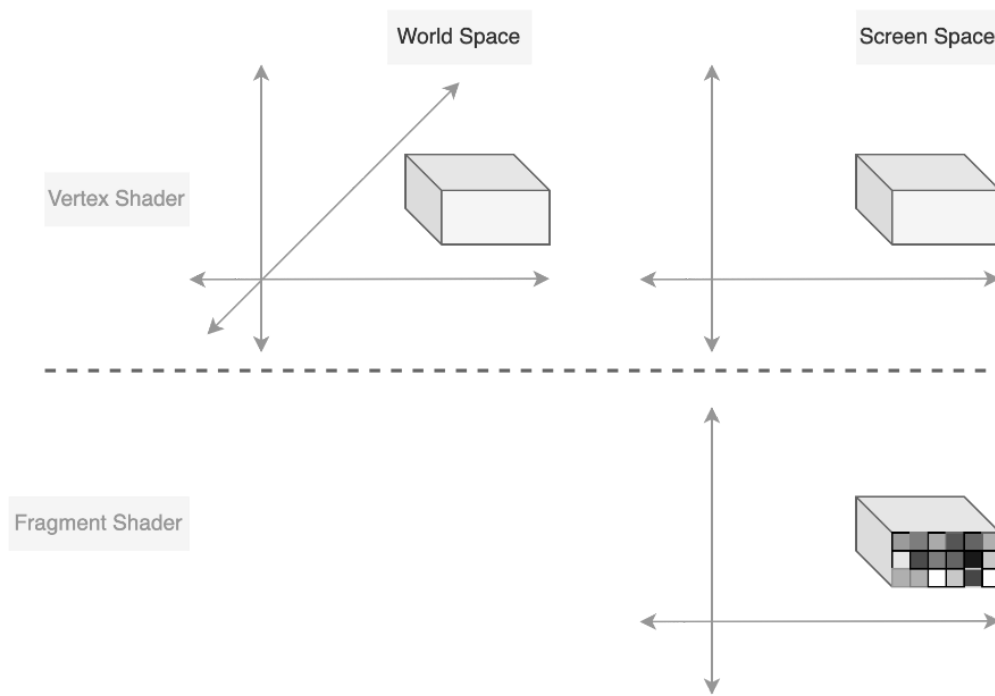
The key point to remember is to think about the attribute or uniform or varying as what provides the uniqueness for every pixel or shader run in fragment shaders and vertex shaders (and compute shaders or any type of shader).

What do I mean?

The inputs provide uniqueness for each shader run or each red square above. So when all the red squares run at the same time, each red square will have a slightly different output (`gl_Position` or `gl_FragColor` the corresponding built in outputs for GLSL shaders), which means a slightly different 3D position in the vertex shader or a slightly different color in the fragment shader.

This leads nicely onto what shaders run first and what is the order of things? I love visualizing concepts. When you can write something down or visualize it, 90% of the problem and / or solution becomes apparent. (For me anyway, I encourage you to find what works for you).

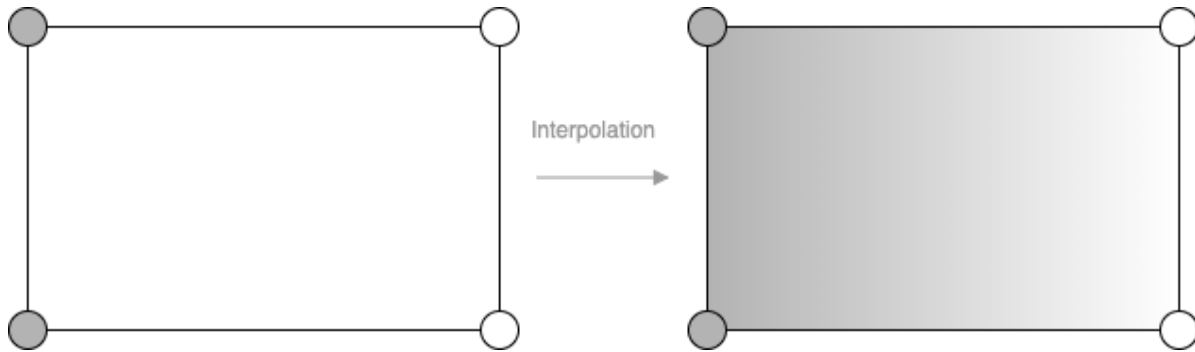
Consider this basic diagram which outlines the processes in vertex and fragment shaders:



The vertex shader runs first and runs for every vertex being processed in the geometries of the scene. Then the fragment shader runs, between vertices there might be 200 pixels so how do we get data for each of these fragments if we only have say 4 vertices spread out in a square shape on the screen?

One of the mind bending things to realize is if vertices are processed by vertex shaders and pixels are processed in fragment shaders. So how do I go from 7 vertices above to 18 fragments or pixels? (This is a gross under simplification, but it helps illustrate the concept for people not used to it). So we now understand the ratio is not 1:1 so how do we go from for example 1:100 vertices to fragments being processed and use unique data from uniforms and varyings?

Here is the magical thing: interpolation! Well not so magical and more mathematical. Built into the pipeline between vertex and fragment runs any data going from the vertex shader to the fragment shader will be interpolated. A way to think of this is this:



Interpolation means - is a mathematical technique used to estimate unknown values that fall within the range of known values. In the context of graphics and GLSL, interpolation refers to the process of determining intermediate values for variables across a surface, line, or space.

(Directly from our favorite AI chatbot)

In layman's terms its gradients of data between fixed points, demonstrated above, 4 fixed data points in the vertex shader, 2 red, 2 green and then interpolated into many data points, or a gradient of color / data in the fragment shader. Which creates uniqueness i.e. a gradient rather than block colors.

One thing you need to understand or get used too is that color is just data, positional information is just data. You are learning to understand mathematical concepts in the context of data, not just the abstraction of position or color. Another useful thing to

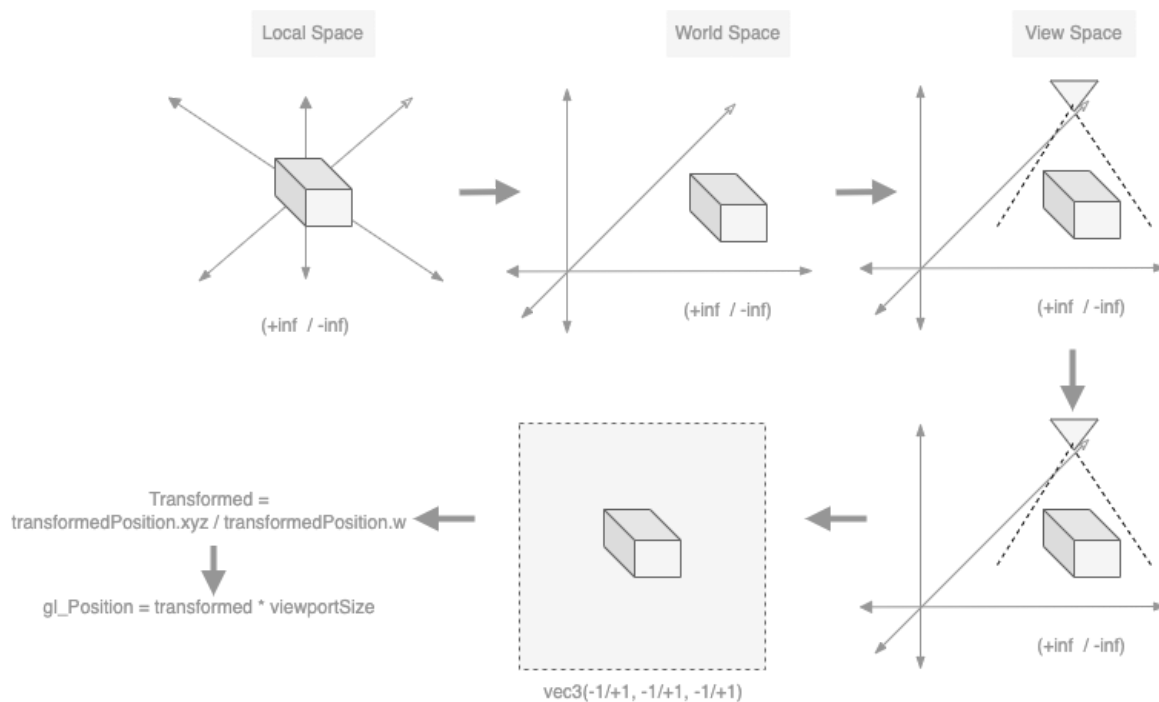
know in approaching shader topics is get to know common ranges of the built in variables or coordinate systems. It is incredibly useful.

Ok so we now know what shaders are, we know the order they run, we know data is interpolated between vertex and fragment shaders and we now need to know how do we go from a 3D space to a 2D space? As demonstrated in the diagram above.

Coordinate systems enter the room.

In a vertex shader matrices are used to transform between coordinate systems. You will often see this: `gl_Position = projectionMatrix * modelViewMatrix * vec4(position, 1.0);`

So consider this diagram:



GLSL API

Types

Types are much like typescript with javascript. Everything has a type. If you forget the type when typing, check the console and inspect the error in red. It will show you exactly where the problem is. The errors in the console can look intimidating at first but it's just like any other stack trace and the great thing is that it will give you a line number. Expect at first there to be a dozen or two errors while you get used to it. It is extremely strict; it complains about missed semicolons at the end of sentences, types and miss defined functions etc.

Scalars

Float's are numbers which have a decimal component like 1.23 and 28.987. Important aspects to this decimal are the fact we can only rely on a certain number of significant digits.

What's a significant digit?

Its easier to explain the opposite first. Anything which goes beyond the significant numbers would be classed as a fuzzy digit, which changes over time and does NOT remain consistent from one frame to the next. Where as there significant digits remain constant or there is such a negligible chance it is essentially constant. This is a limitation of hardware not software. If you're interested I'd recommend reading around the concept of fuzzy numbers in computer science. It's a fascinating topic.

Why is this even important? So imagine we have a very wide range in colors and

vertices. This higher range would result in more data being compacted into a range of $[-1 - +1, -1 - +1]$ clip space for example. This would mean a larger number of digits to the right of the decimal point.

If there's a larger number of digits and we go beyond the precision of GLSL shaders we can not guarantee the digits will be consistent frame by frame. What would this translate too? A loss in detail. So in general the higher the precision the more detail you will have.

Ints are just whole numbers like 1 or 10 and you have the same limitation that only a certain number of significant digits are reliable, Bool's are binary true or false values.

Some examples of scalar types:

```
float myValue = 1.0;
```

```
int myInt = 1;
```

```
bool myBool = true;
```

Vector Types

We know what vectors are, maps to points in 3D space. But how do we define them?

```
vec2 myVec2 = vec2(0.1, 0.1);
```

```
vec3 myVec3 = vec3(0.1, 0.1, 0.1);
```

```
vec4 myVec4 = vec4(0.1, 0.1 0.1, 0.1);
```



The way to access them is as follows:

```
myVec2.xy
```

```
myVec3.xyz
```

```
myVec4.xyzw
```

Or swizzle:

```
myVec4.xxyy
```

```
myVec4.xxyy
```

Swizzling is a way to access the components of a vector. These components can also be called channels and the diagram below gives an example of this.

In the Fragment shaders you might start to realize my point about these channels or components just being data, colors and positional data sharing the same components /

channels. So if we were to output the red channel, it is our meaning we add to it which is important, is it a color or is it representing positional data..

The really bizarre thing is that given experience and time using shaders you start to realize X/Y/Z coordinates have color. The abstraction is a bit hard to understand but a vector is a vessel and the meaning is in what we associate with it, i.e. the color red or the positional component X.

In the fragment shader you need to understand that the min output value in `gl_FragColor` is a value of 0.0 and the max values are 1.0. So if you try and do something below into the minus numbers or do something which results in a value over 1, It wont work and will clamp at either 0 or 1. This is just how the shader works, there's no way around this. And is often a source of confusion when first starting out as to why calculations or colors aren't what you expect.

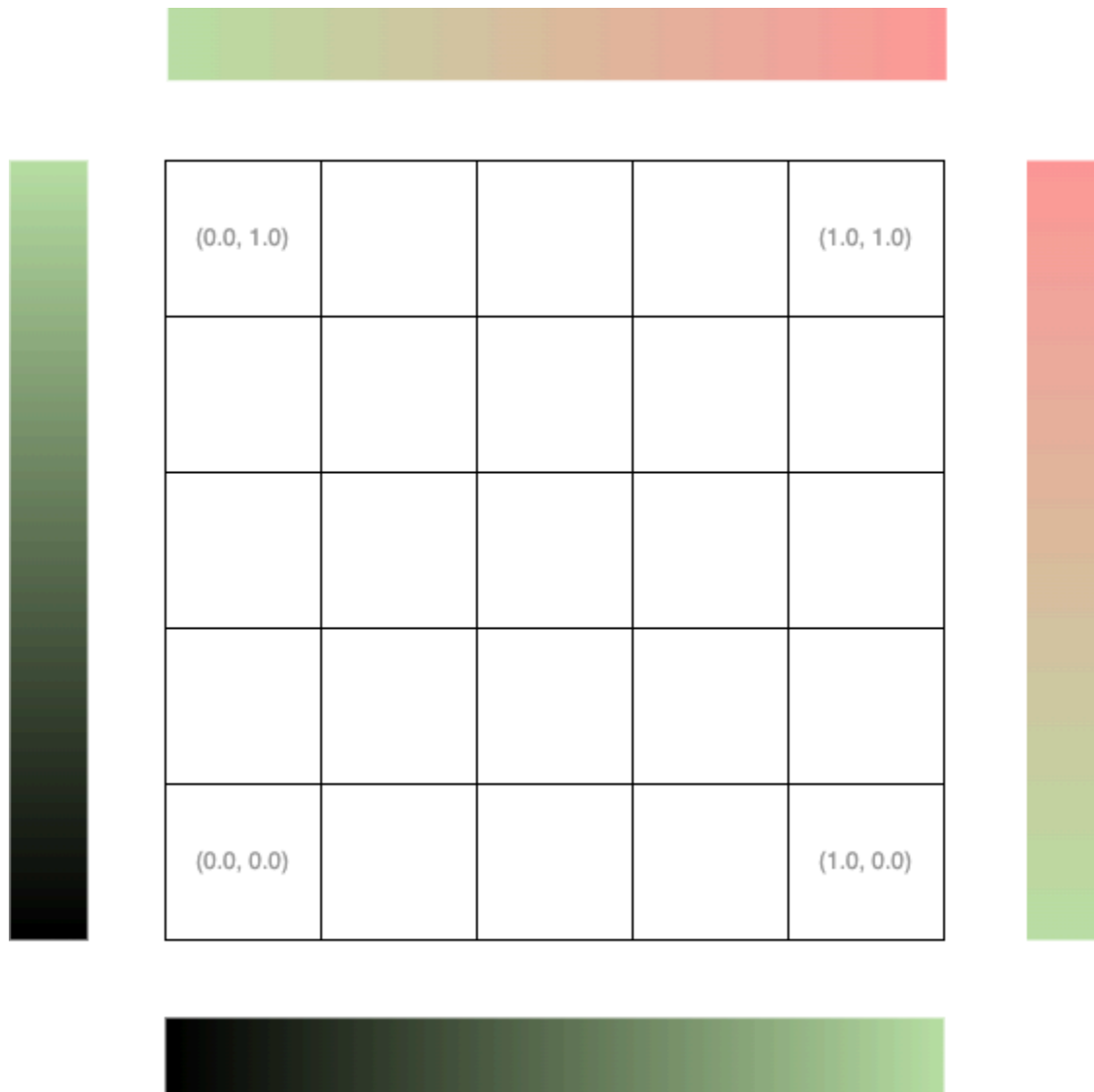
Matrix types

Textures, Mapping and Sampler Types

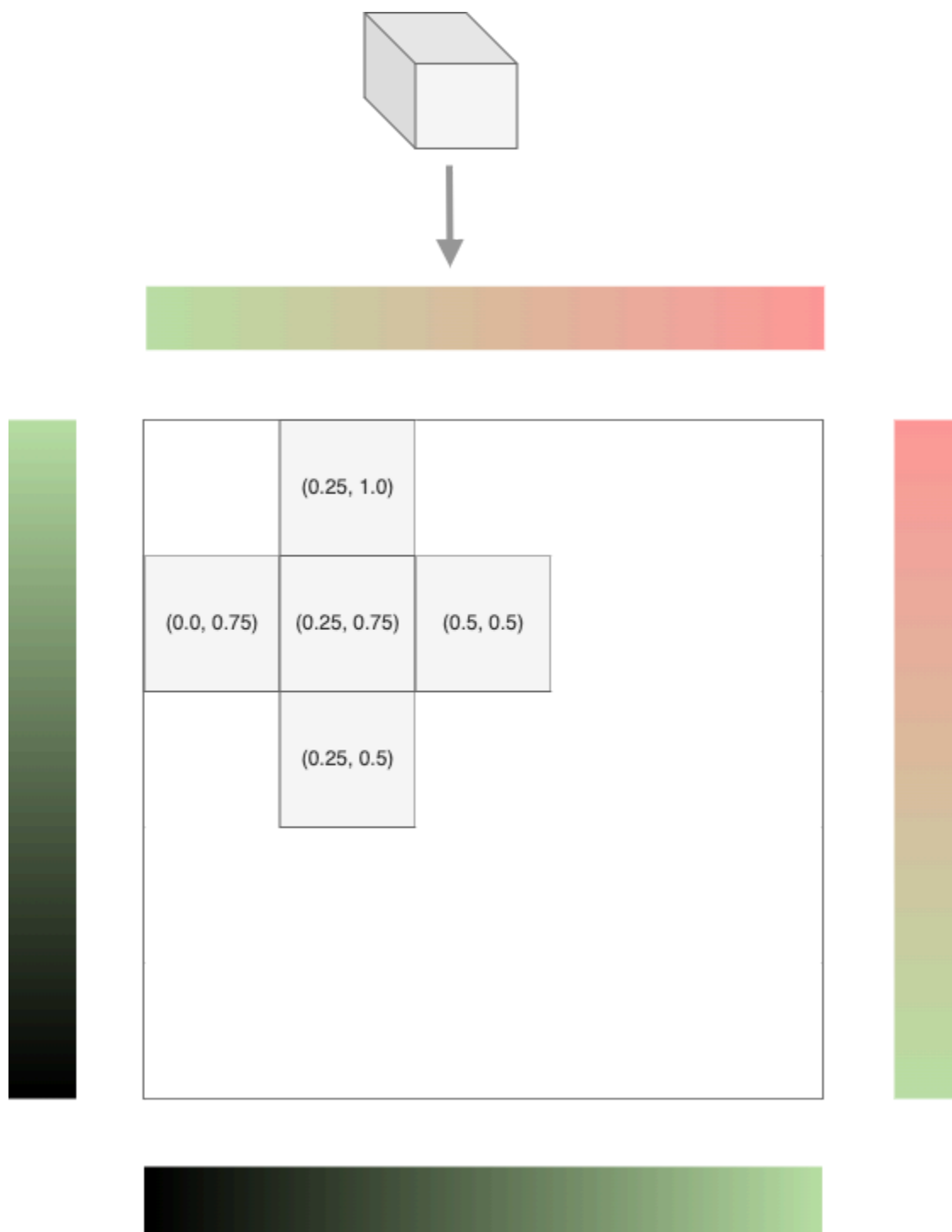
Textures are a way of transferring data and consuming data on the GPU. A texture is just another way of expressing an image. It's a specific format loaded from image loaders, which gives the data in a set way for the gpu to consume.

How do we access this texture in a highly parallel GPU environment?

We can sample the texture using UV coordinates. UV coordinates are as follows:



Above is a uv map, it goes from 0,0 to 1,1 and vertices are unwrapped onto this map like so, we map points of the 3D object to the UV map and the Textures can be aligned with the uv map:



So it's usually per vertex for the UVs but the diagram above is just symbolic that every vertex will have a UV coordinate and then they will be interpolated from the vertex to fragment shaders. All per vertex attributes and vertices themselves will have all the data stored in typed arrays.

So where do textures come in? They are usually aligned in 3D software and then the islands on the UV map will have specific colors.

Types

GLSL and any shading language are heavily typed, if you come from the TS (TypeScript) world, you should feel right at home. It is tricky to start with as the compiler is very strict i.e. a missing semicolon and the compiler throws an error. Shader languages are low level and closer to the binary than a higher level language like JavaScript. You'll notice the syntax is very different in that you won't have all the whistles and bells than JavaScript has. It is very similar to rust and has a lot of the same concepts.

Everything you write in GLSL (the THREE.js shader language of choice) has to have a type:

```
varying vec2 vUv;

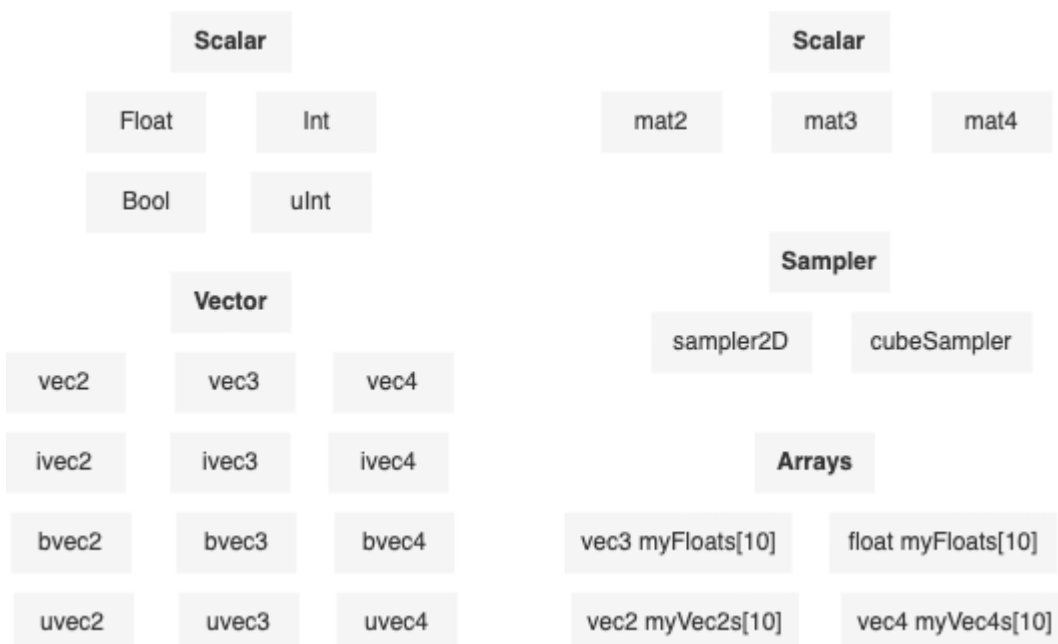
uniform sampler2D tex;

main () {
    vec3 color = texture(tex, vUv);

    gl_FragColor = vec4(color, 1.0);
}
```

Some of the main types are listed below, this can range from scalars, to matrices, to vectors, to samplers and arrays. The interesting thing about arrays is that you have to define the memory or the length of the array upfront, this is a requirement for the low level nature of GLSL, you have to define the length to allocate fixed memory. You cannot have dynamic arrays. If you use less than what you define that's fine but you cannot exceed the length you define. The maximum capacity of an defined array will be limited by the GPU capacity of your machine , unique for different GPUs.

As a side point, although it shows a variety of vectors and matrices, it should be noted that in math you can have say a vec8 or mat5, its just a limitation of this computer language the size of them.



Control Flow

Control flow is essentially if else statements you find in JS. It is also referred to as branching:

```
if (condition) {  
  // Code executed if condition is true  
  
} else if (other_condition) {  
  
  // Code executed if other_condition is true  
  
} else {  
  
  // Code executed if none of the above conditions are true  
}
```

Mathematical Functions

Two useful functions are degrees and radians, which allow for the conversion between degrees and radians. I don't use these an awful lot but it's important to differentiate between the two types of angles and not to get mixed up.

`radians(float degrees)` — Converts degrees to radians.

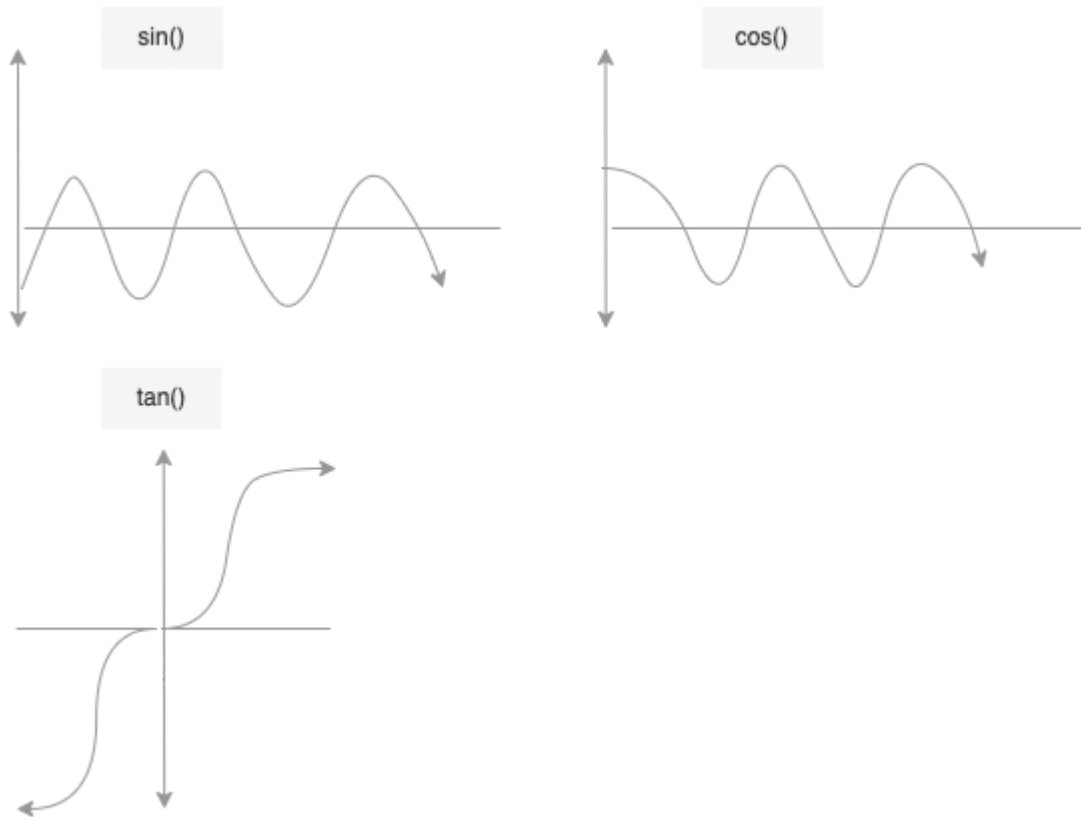
`degrees(float radians)` — Converts radians to degrees.



`sin(float angle)` — Computes the sine of an angle.

`cos(float angle)` — Computes the cosine of an angle.

`tan(float angle)` — Computes the tangent of an angle.



Sometimes you will want to apply a math function like the ones below to generate particular shapes or patterns, these are not the common ones I have ever used though. Still can come in useful sometimes.

`asin(float x)` — Computes the inverse sine (arcsin) of x .

`acos(float x)` — Computes the inverse cosine (arccos) of x .

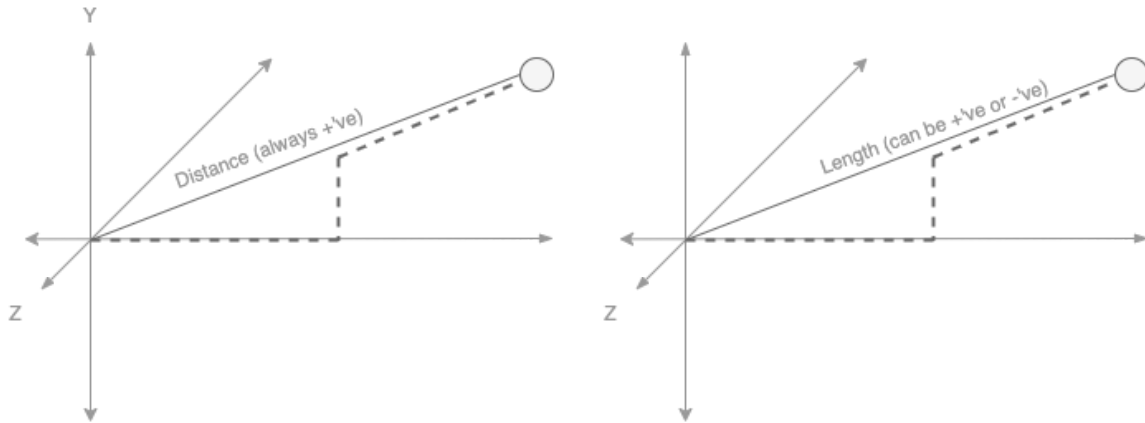
`atan(float y, float x)` — Computes the arctangent (`atan2`) of y/x .

`atan(float x)` — Computes the arctangent of x .

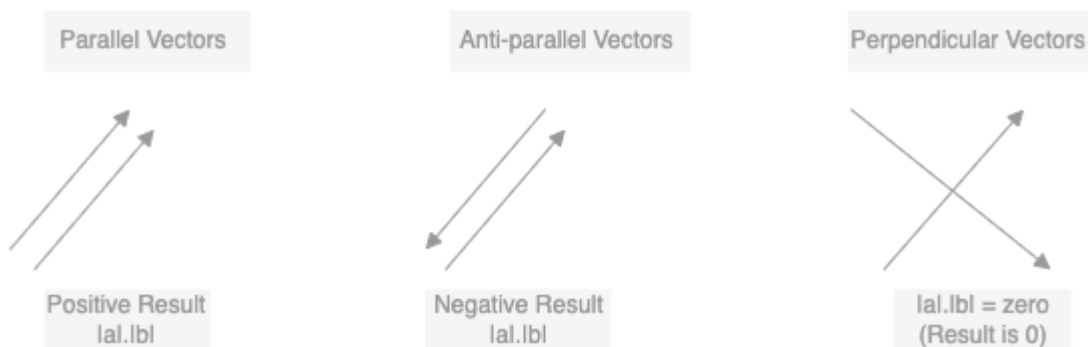
Geometric Functions

`distance(p1, p2)` - Computes the distance between two points.

`length(v)` - Returns the magnitude (length) of a vector.

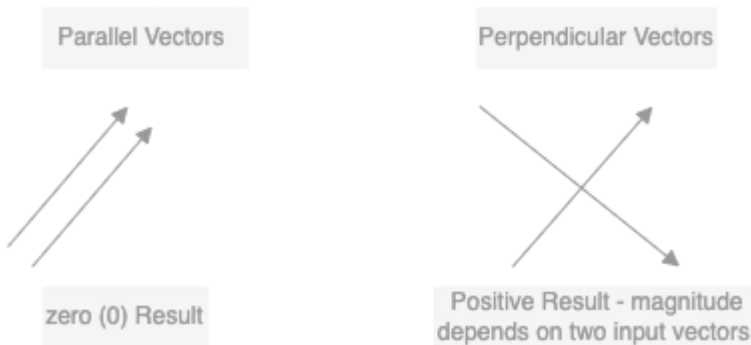


`dot(v1, v2)` - Computes the dot product of two vectors.



You will often hear the dot product is used to see how much vector A is projected onto vector B i.e. how parallel are the vectors? The result and the sign of the result are then used to describe the relationship between the two vectors.

`cross(v1, v2)` - Computes the cross product of two 3D vectors.



```
// code example
vec3 AB = v1 - v0;
vec3 CB = v2 - v0;

vec3 surfaceNormal = cross(AB, CB);
```

`normalize(v)` - Returns a unit vector in the same direction as the input vector. It is important to realise that unit vectors or direction vectors don't conventionally have magnitudes; they are just that, a direction. An important concept when studying or building things in GLSL. This concept is one you really will need to understand.

```
// Normalising a vector
vec3 vector = vec3(1.0, 1.0, 1.0);

float magnitude = sqrt(vector.x * vector.x + vector.y * vector.y +
vector.z * vector.z);

vec3 normalisedVector = vector.xyz / magnitude;

or/

vec3 vector = vec3(1.0, 1.0, 1.0);

float length = length(vector);

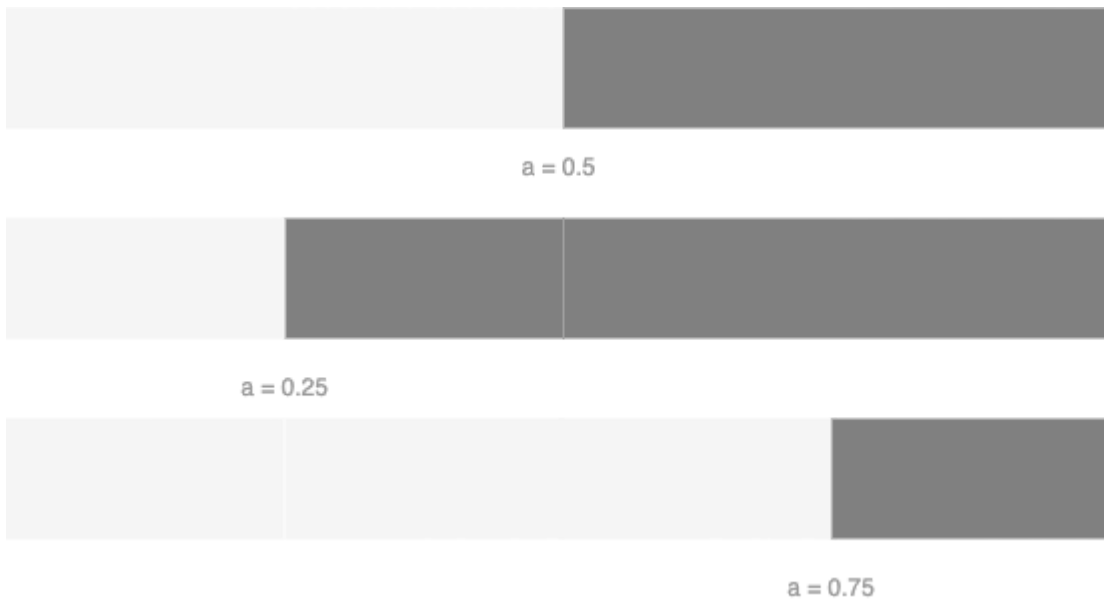
vec3 normalisedVector = vector.xyz / length;
```

`reflect(l, N)` - Computes the reflection direction for an incident vector.

`refract(l, N, eta)` - Computes the refraction direction through a surface.

`faceforward(N, l, Nref)` - Adjusts the normal to face the correct direction.

`mix(x, y, a)` - Linearly interpolates between two values.



`clamp(x, minVal, maxVal)` - Restricts a value to a given range.



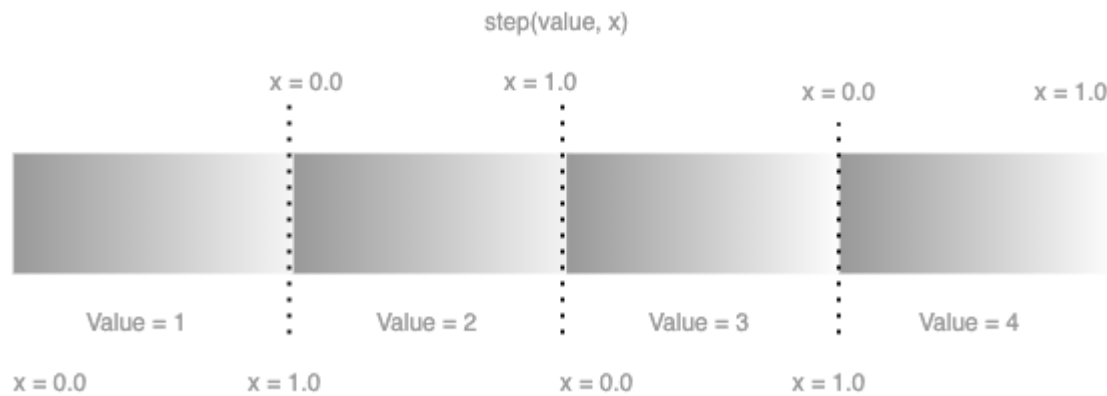
`step(edge, x)` - Generates a step function between 0 and 1.



`smoothstep(edge0, edge1, x)` - Smoothly interpolates between 0 and 1.



`mod(x, y)` - Returns the remainder of division (modulo).



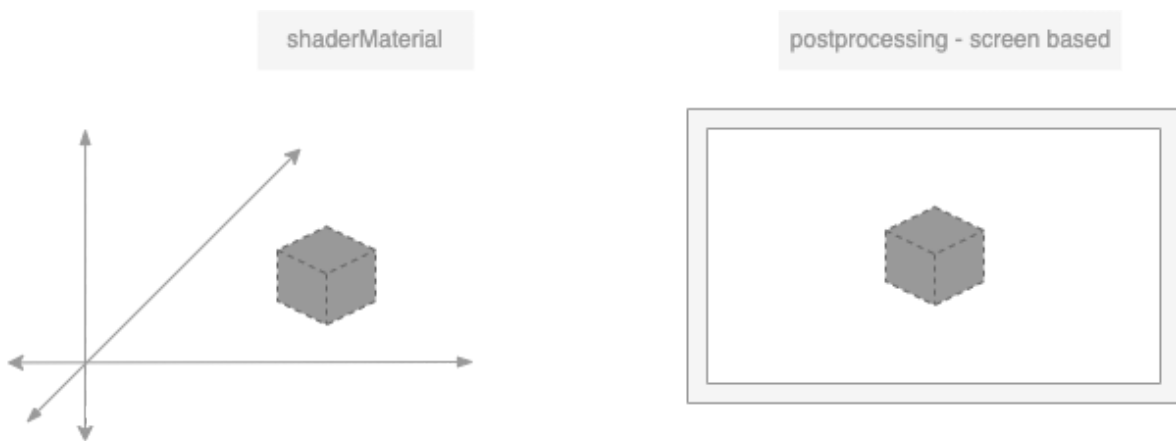
`transpose(m)` - Returns the transpose of a matrix.

`inverse(m)` - Returns the inverse of a matrix.

Material and Post Processing Shaders

Materials are what give the look of a geometry, if the geometry provides the shape the material gives the look. Postprocessing is where effects and techniques are used on the entire screen and not just one material on one geometry in a 3D world.

One is confined to the shape of a 3D geometry and one is applied to the entire screen or canvas viewport.



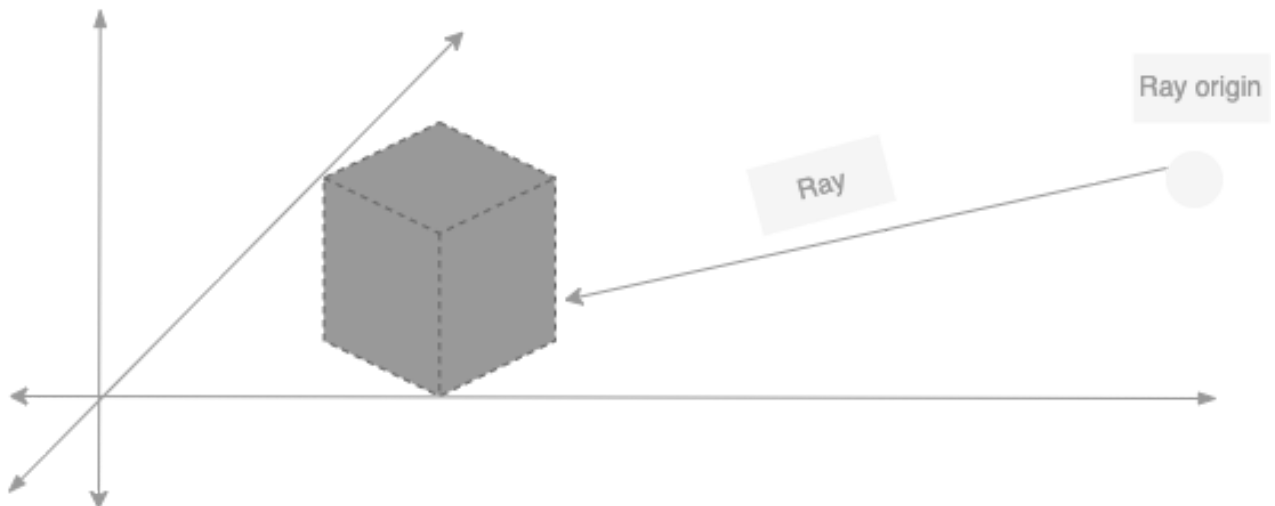
GLTF

GLTF is the format of choice for 3D models on the web. It stores the links to the textures, per vertex attributes like UV's, normals and vertices in a convenient format. It is well supported and has loaders in three.js. GLB and GLTF are different formats of the same thing. It is loaded in the same way in THREE.js.

Useful Concepts

Raycaster

A raycaster is a unique way to examine what is in front of a point in space. In 3D space we don't have mouse events, we have to mathematically put a point in 3D space and cast a ray from it, every time the raycaster increases in length over time we check if it has hit a face of a mesh. When we get an intersection we will gather a lot of information like the face and point on that face which intersected.



Hulls

Concave & Convex Hulls

Hulls in most three.js contexts are perimeters around a set of points, Whether the perimeter is tight or loose depends on if we use concave hull or convex hull respectively. Below gives a visual representation of the hulls. You can use libraries like hull.js or similar in JavaScript. Usually accepting x/y coordinates and spitting out the perimeter coordinates:



Once we have the hull what can we do with it? Well three.js allows you to make shapes like so:

```
// 2D Set of Points
const shape = new THREE.Shape().setFromPoints(points);

//Geometry
const shapeGeometry = new THREE.ShapeGeometry(shape);
```

Barycentric Coordinates

Barycentric coordinates are used for things like wireframes. It can essentially tell you if a coordinate is in or on individual triangles, or, can be used as an edge detection for multiple triangles with some additional logic. For the single triangles you could do:

```
float edgeWidth = 0.02;

float edgeDistance = min(min(vBarycentric.x, vBarycentric.y),
vBarycentric.z);

float wireframe = smoothstep(0.0, edgeWidth, edgeDistance);
```

dFdx & dFdy - GLSL partial function

The partial derivative function in essence checks the neighboring pixels and returns the color for it. Why would this be useful? Well if we have the neighboring color we can detect a change from one pixel to its neighbor, so we can detect gradients of change.

Ok great so we can detect gradients of change but what can they be used for? Edge detection and bump maps are to name but a couple of things.



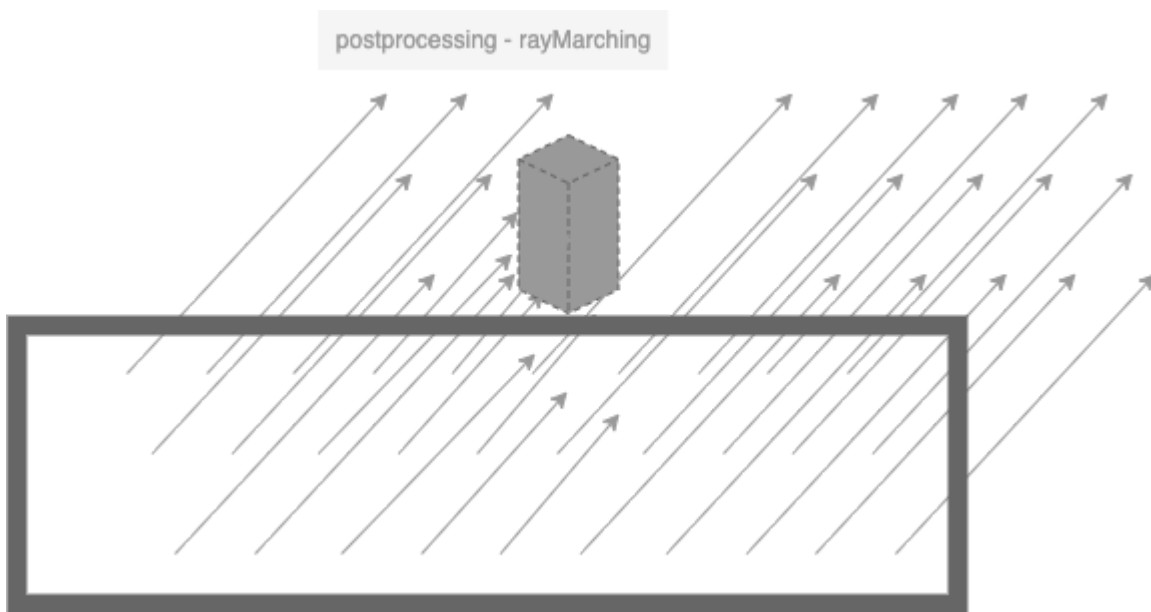
```
uniform float threshold;
```

```
void main() {  
    // sample neighbouring vertical and horizontal vertices  
    vec3 dFdxColor = dFdx(fragColor);  
    vec3 dFdyColor = dFdy(fragColor);  
  
    float colorChange = length(dFdxColor) + length(dFdyColor);  
  
    if (colorChange > threshold) {  
        gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);  
    } else {  
        gl_FragColor = vec4(fragColor, 1.0);  
    }  
}
```

In the example above we check the neighboring fragments and then check if the strength of the color is above a threshold. I.e. does the sampled color change or a certain gradient between fragments reach a threshold. If it does then do some logic in a branch.

RayMarch Algorithm

RayMarching is a technique used in shaders to start to define a 3D world. How does it do this? Usually by SDFs, there are some other more experimental ways. So imagine that from the camera every pixel of fragment has a ray fired out from it and then at each run of the shader we check to see if the tip of these rays hits inside of an SDF function. And then set the colors depending on if inside or outside the SDF shape. Some of the rays will not hit anything but some will be demonstrated below.



Compute shaders

Compute shaders are used for heavy computing tasks, imagine we had 10 layers of noise we had to calculate, ideally we would use compute shaders as they are best suited for heavy calculations.

Array Uniforms

Arrays are defined in shaders with fixed length, you have to pre allocate memory, you cannot dynamically change arrays. In addition to this, arrays will have a max capacity or size depending on hardware. This isn't to say you can't use template literals in JS and define a fixed length. This can be tricky to set up though.

This is a very good example of hardware restrictions with GPU's. There will usually be 1000 or 2000 items in an array. You will want to be aware of this and not go close to the max capacity. Below is a way you would define the array as a uniform:

```
// Examples of defining various array with fixed length and memory
uniform vec2 myVec2Array[10]
uniform vec3 myVec3Array[10]
uniform vec4 myVec4Array[10]
uniform int myIntArray[10]
uniform float myFloatArray[10]
```

Generating UV's Manually

If we have a 2D polygon shape we can generate new UVs approximately by utilizing the bounding box and normalizing the position attribute values. This allows us to set new UVs if we cannot set them in 3D software and we have created a shape dynamically.

```
//Example of generating crude UVs for a 2D geometry
geometry.computeBoundingBox();
const bbox = geometry.boundingBox;

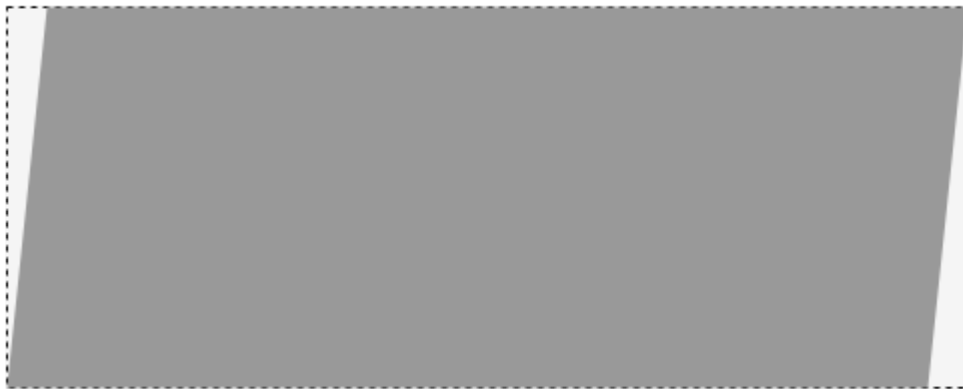
const min = bbox.min;
const max = bbox.max;

const uvs = [];
const positionAttribute = geometry.getAttribute('position');
for (let i = 0; i < positionAttribute.count; i++) {
    const x = positionAttribute.getX(i);
    const y = positionAttribute.getY(i);

    const u = (x - min.x) / (max.x - min.x);
    const v = (y - min.y) / (max.y - min.y);

    uvs.push(u, v);
}
geometry.setAttribute('uv', new THREE.Float32BufferAttribute(uvs, 2));
```

Max y Bounding Box



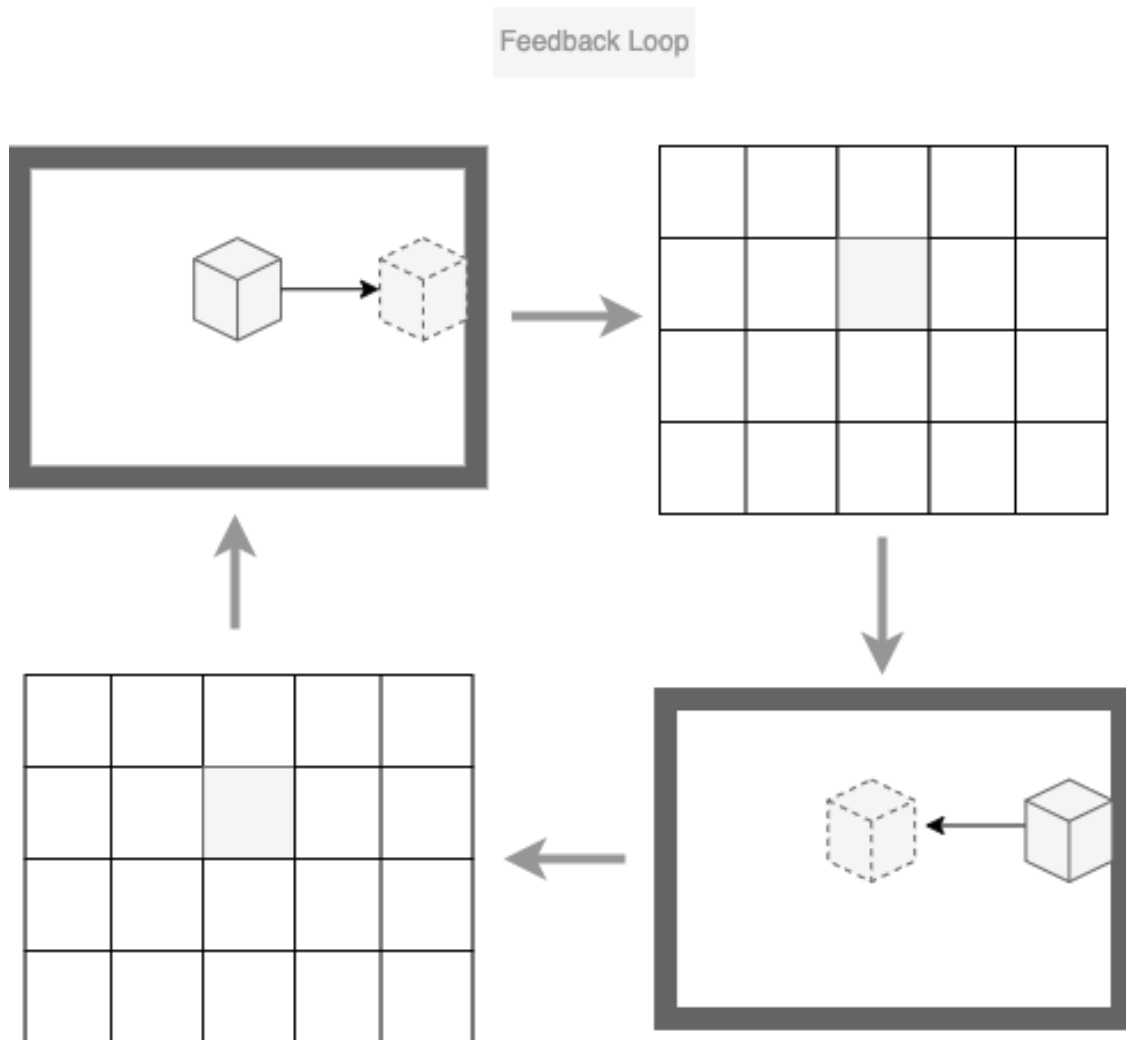
Min x/y Bounding Box

Max x Bounding Box

Texture Feedback

Texture feedback is the backbone of many advanced rendering patterns. It is a sort of abstract way of keeping state. Although this seems like an oxymoron - as GPUs are

stateless. It is a way of passing the previous state to the current frames inputs. You essentially use render targets to render the scene too and process in the current frame. The way to visualise this is shown below:



A ping pong of the textures is because WebGL cannot have the input buffer as the output buffer directly, but you can ping pong the textures i.e. switch them around, so the input becomes the output and the output becomes the input buffer.

An example of a texture feedback or ping pong (as you will probably see on the internet) is as follows:

```
const offScreenFBOTexture = useFBO(1024, 1024, {
  minFilter: THREE.LinearFilter,
  magFilter: THREE.NearestFilter
});
const onScreenFBOTexture = useFBO(1024, 1024, {
  minFilter: THREE.LinearFilter,
  magFilter: THREE.NearestFilter
});

let textureA = offScreenFBOTexture;
let textureB = onScreenFBOTexture;

useFrame(({ gl, camera }) => {
  gl.setRenderTarget(textureB);

  //Swap textureA and B
  var t = textureA;
  textureA = textureB;
  textureB = t;
  gl.setRenderTarget(null);
  gl.render(scene, camera);
});
```

This foundational knowledge allows you to do more advanced techniques like GPU particles or other effects like wakes or smoke-like effects spreading on the screen or other simulations.

OnBeforeCompile & Existing Materials

In most circumstances which involve lighting or custom vertex manipulation, you probably want to opt for modifying an existing material and not do all the calculations

for light yourself. It is complicated and a very steep learning curve. So `onBeforeCompile` on a `MeshStandardMaterial` for example will be a much preferable option. It essentially allows you to modify existing materials before the shader is compiled. This means we can add custom code to the existing material. Allowing you to get all the benefits of the prebuilt materials, but also adding in your own effects.

```
// onBeforeCompile callback function
const OBC = useCallback((shader) => {
  shader.uniforms = { ...shader.uniforms, ...customShader.uniforms };

  shader.vertexShader = shader.vertexShader.replace(
    "#define STANDARD",
    customShader.vertexShader.top
  );
  shader.vertexShader = shader.vertexShader.replace(
    `#include <fog_vertex>`,
    customShader.vertexShader.bottom
  );
}, []);

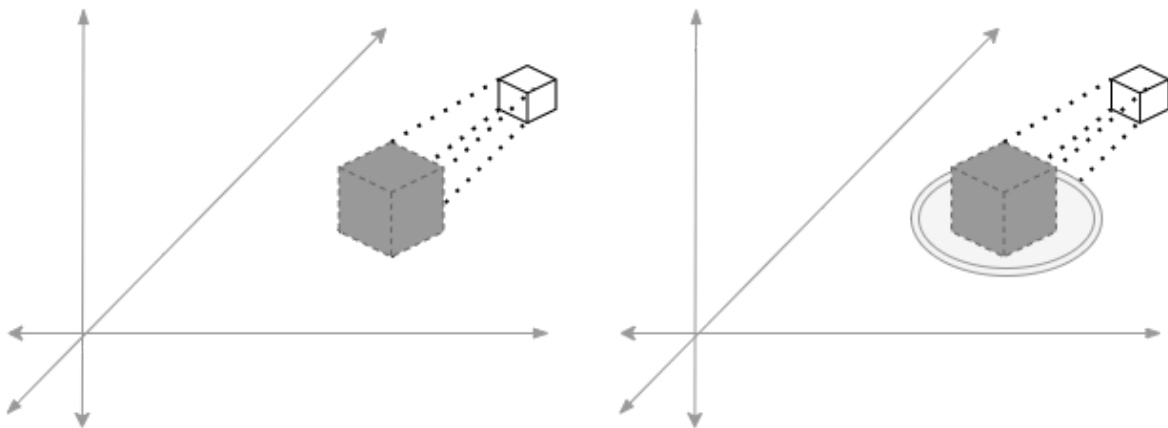
// Use in a declarative way
<meshStandardMaterial
  attach="material"
  onBeforeCompile={OBC}
/>
```

The callback allows you to replace sections of code with custom code. Remembering you have to add in the replaced code again so you don't remove any essential material code in the THREE.js library. There are ways to update the uniforms as long as you spread the uniforms in this way you can then access your custom shader and update the uniforms in the `useFrame`.

Case Studies

Depth textures, world coordinates and postprocessing

This is a small interesting experiment which goes through several important features of shaders and three.js. We will explore reversing the coordinate system, depth textures and introduce SDF's. The aim of this experiment was to examine how to use depth to produce world coordinates and then use an SDF to visualize it. Here's an overview:



Depth and depth textures are relative to the camera, it states that the fragment we are processing has an color from an object from the 3D world in it and has depth shown by blacker values, i.e. less color, less data.

```
// White === max data in fragment shader output
gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);

// Black === min data in fragment shader output
gl_FragColor = vec4(0.0, 0.0, 0.0, 0.0);
```

In some circumstances you can initialize a vec4 like this also , which is entirely valid:

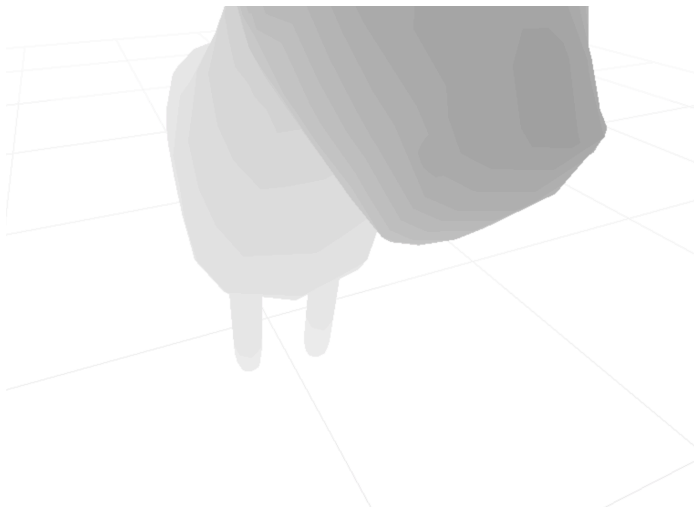
```
// Initializing a vector shorthand
```

```
vec4 myValue = vec4(1.0);
```

Back to the task at hand; the depth is stored in the x channel of the depth texture.

When I say depth texture I just mean a rendertarget with specific settings which allows us to extract a depth texture which stores data in the r or x channel.

The darker the color i.e. the blacker it is, the closer the object in the 3D world is to the camera. A typical depth texture set to display in post processing looks like this:



The way we would display the depth in the post-processing shaders is by sampling the depth texture and then displaying it in every channel of the output `gl_FragColor`. This will give us a grayscale image as seen above.

```
// Vertex Shader
varying vec2 vUv;

void main () {
    vUv = uv;
    gl_Position = projectionMatrix * modelViewMatrix * vec4(position, 1.0);
}
```

```

// Fragment Shader
uniform sampler2D depthTexture;
varying vec2 vUv;

void main () {

    vec4 depthColor = texture(depthTexture, vUv);

    gl_FragColor = vec4(depthColor.x, depthColor.x, depthColor.x, 1.0);

}

```

A couple of key things to talk about here is the `vUv` varying and the `depthColor`. Varyings are defined from the vertex shader to the fragment shader and they then are interpolated as described earlier. Above is the pattern for passing this data between shaders.

When we sample a texture in post processing, each fragment we process or each time our code is run in the parallel environment will have a unique set of UV coordinates.

Remember uv's are the coordinates we use to sample textures, in this sense the texture is a snapshot of the screen of the canvas viewport, storing all the data of the scene into the colors of a rendertarget and allowing us to manipulate it.

This is a key concept to understand if world coordinates and materials are everything in a scene in a mathematical model, then the camera and post processing shaders facilitate a snapshot of the colors; by default the position of the colors on this rendertarget show or indicate positional information as well, after being transformed by going through all the coordinate systems as described earlier.

A side note is the UV's are inbuilt in the vertex shader and is why we can just use the reserved letters uv without defining anything before it. If you examine the post-processing shaders in the three.js repo, you will find a lot of built in variables and functions you can use for various tasks, uv's being one of them. The point demonstrated below and is something to remember when using the three.js shaders, is this:

```
// A lot of built in three.js code
// (We do not see this)
// (we can access variables though)
//
// A lot of built in three.js code
////////////////////////////////////
////////////////////////////////////

void main () {

}
```

RawShaderMaterials in three.js are the opposite of this - they allow for the use of shaders but stripping most of the builtin variables, this can be incredibly useful for fine grain control of what you are doing in the GLSL shaders in three.js, or if you are doing some kind of linting or prettier setup for the GLSL code in your code editor of choice.

Once the depth has been established relative to the camera, how can we get the 3D world coordinates from it? And with this use an SDF without any raymarching (a technique to define objects in a post-processing environment).

The function below goes through the process:

```

vec3 worldCoordinatesFromDepth(float depth) {
    float z = depth * 2.0 - 1.0;
    vec4 clipSpaceCoordinate = vec4(vUv * 2.0 - 1.0, z, 1.0);
    vec4 viewSpaceCoordinate = projectionMatrixInverse *
    clipSpaceCoordinate;

    viewSpaceCoordinate /= viewSpaceCoordinate.w;

    vec4 worldSpaceCoordinates = viewMatrixInverse *
    viewSpaceCoordinate;

    return worldSpaceCoordinates.xyz;
}

```

In the context of extracting data from float textures, values will be clamped to 0 and 1 for each channel of a vector inside the GLSL shader code.

This example is jumping into clip space (which ranges from -1 to 1 on every axis). A common way to transform something from 0 to 1 to -1 to 1, is to carry out this transformation, i have given the extremes as the example:

```

float range0 = 0.0; // range 0-1
float range1 = 1.0; // range 0-1

float newValue0 = range0 * 2.0 - 1.0; // range -1 - 1
float newValue0 = -1.0;

float newValue1 = range1 * 2.0 - 1.0; // range -1 - 1
float newValue1 = 1.0;

```

The process is simple. We want to reverse the coordinate systems to get to the world coordinates again. And then we can use Our SDF to visualize the world coordinates. So symbolically going through the process see below, starting at screen coordinates and going back to world coordinates.



So if we have all the information of the forward process to go from 3D to 2D. We can establish a backward process to reverse it. It's important to realize we have irretrievable data loss going through the forward process. So we can and will reverse this forward process however it won't be 100% accurate. Still a nice introduction to learning about these systems. And in some circumstances very useful.

When we want to go from clip space to view space we need the inverse of the projection matrix as we are going backward.

```
vec4 viewSpaceCoordinate = projectionMatrixInverse * clipSpaceCoordinate;
```

To then go from viewSpace to world space we need to do the reverse of the perspective divide and then multiply by the inverse of the view matrix as it is the backward process.

```
viewSpaceCoordinate /= viewSpaceCoordinate.w;
```

```
vec4 worldSpaceCoordinates = viewMatrixInverse * viewSpaceCoordinate;
```

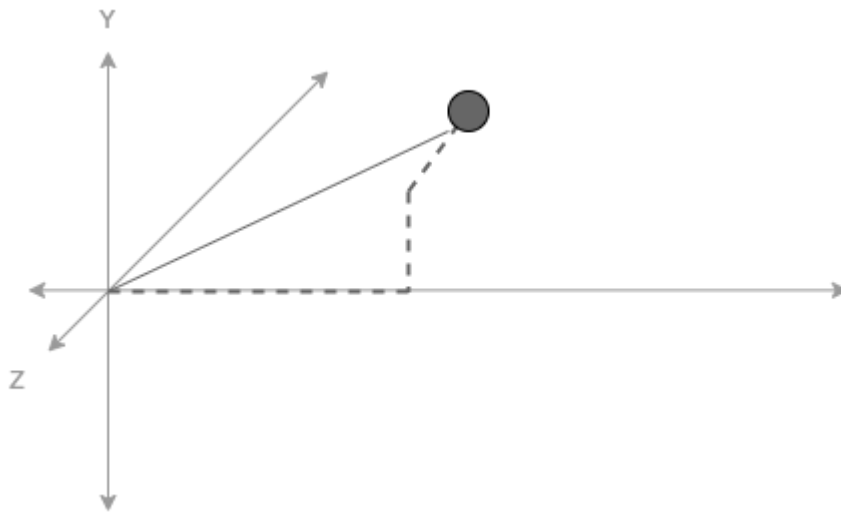
Ok we now have the coordinates in world space, which is just local space with all of the 3d transforms (rotation, scale and translation) moving the geometry away from the local origin and placing it accurately in the 3d scene. What do we want to do with it now?

SDFs are signed distance fields which allow you to mathematically create shapes. I encourage you to read up on SDFs. There are some great resources which will pop up straight away in searching for these SDFs. There are SDFs for all the common shapes. They will have 3D and 2D variations.

SDF's are a mathematical function which gives an output. The sign (+/-) of this output is very important. If the output of this mathematical function is negative then the vector we use as an input to the function is inside the shape and if it's positive then it lies outside of the shape. All SDFs are centered around the origin `vec4(0.0, 0.0, 0.0, 0.0)`, if you want to center an SDF around another point in 3D space you have to transform the input coordinates, there's no need to modify the equation, just the input vector or vectors. This is an example of a spherical SDF:

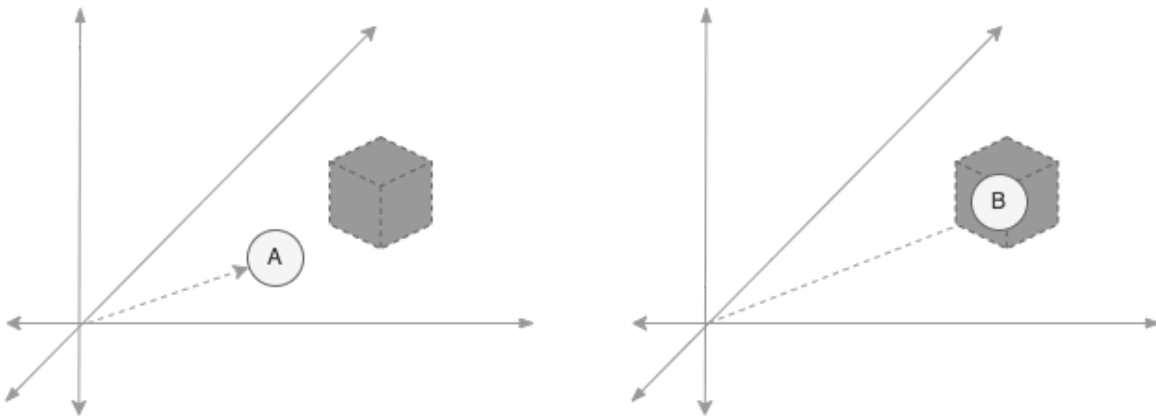
```
float sphereSDF(vec3 p, float radius) {  
    return length(p) - radius;  
}
```

There is lots going on here so let's unpack it. Firstly `length` provides the length of a point from the origin, which is `vec4(0.0, 0.0, 0.0, 0.0)`. Remember the vector is a map of the 3 axes to get to a point in 3D space and the length is then the shortest path from the origin to this point in 3D space:



This can be negative or positive. An important distinction between the length and distance builtin math functions is this, length can be negative or positive and distance provides an absolute value between point A and point B and length provides the distance from or to the origin whichever way round you look at it.

Ok enough minute detail. Why are SDF functions useful? Well if we have a distance from the origin to point A which lies outside of the cuboid SDF, then we have a positive result and if we say, we increase the magnitude of this vector and check every iteration if the point lies within the cube (via utilizing the cuboid SDF). We can eventually obtain a negative value from the SDF or zero if the point lies directly on the cuboid surface - remembering that there isn't a real surface in the truest sense, it's the boundary of the SDF.



Imagine the green represents the radius and point A is equal or less than the radius.

Then this will be true:

```
float radius = 1.0;
float aLength = 0.8;
float bLength = 4.5;

aLength - radius = -0.2; // negative = inside sphere
bLength - radius = 3.5; // positive = outside sphere
```

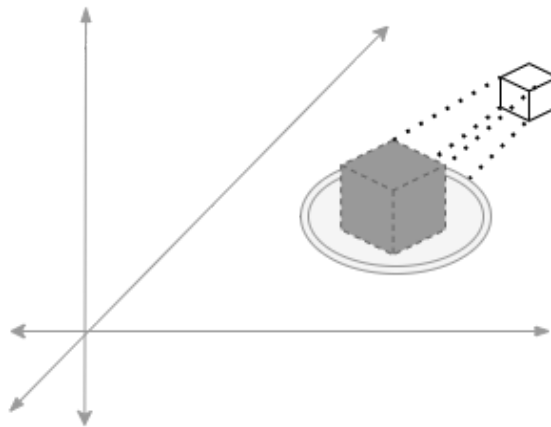
We spoke about modulating values earlier and here we want to modulate the radius i.e. make it pulsate outwards over time, over and over again. So the below does this:

```
float radius = mod(0.1 * time * 10.0, 3.0);
```

To speed up time we want it to go faster so we multiply by a couple of values, 0.1 and 10.0. This speeds time up and makes the effect more interesting. Modulus for anyone who has forgotten does this:



Then the final step is to create something like the rings of jupiter in terms of math and SDF's like a hollow sphere expanding:



The key to doing this is control flows or if else statements, check if the worldPosition is in the SDF and then do another check to see if the worldPosition is not in a smaller radius SDF:

```
if (sphereSDF(worldPosition, radius) < 0.0 && sphereSDF(worldPosition, radius)
> -1.0) {
    gl_FragColor = vec4(0.0,1.0,0.0,1.0);
} else {
    vec3 sceneColor = texture(tDiffuse, vUv).xyz;
    gl_FragColor = vec4(sceneColor, 1.0);
}
```

All the code to this case study can be found in the appendix. (1.0)

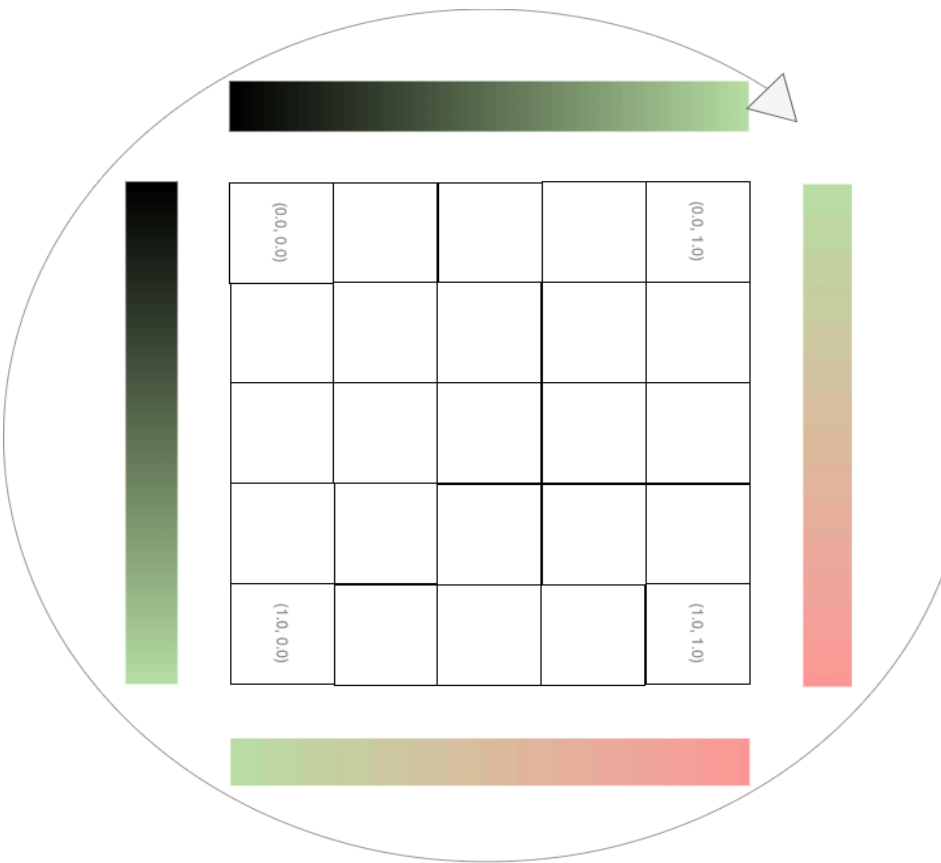
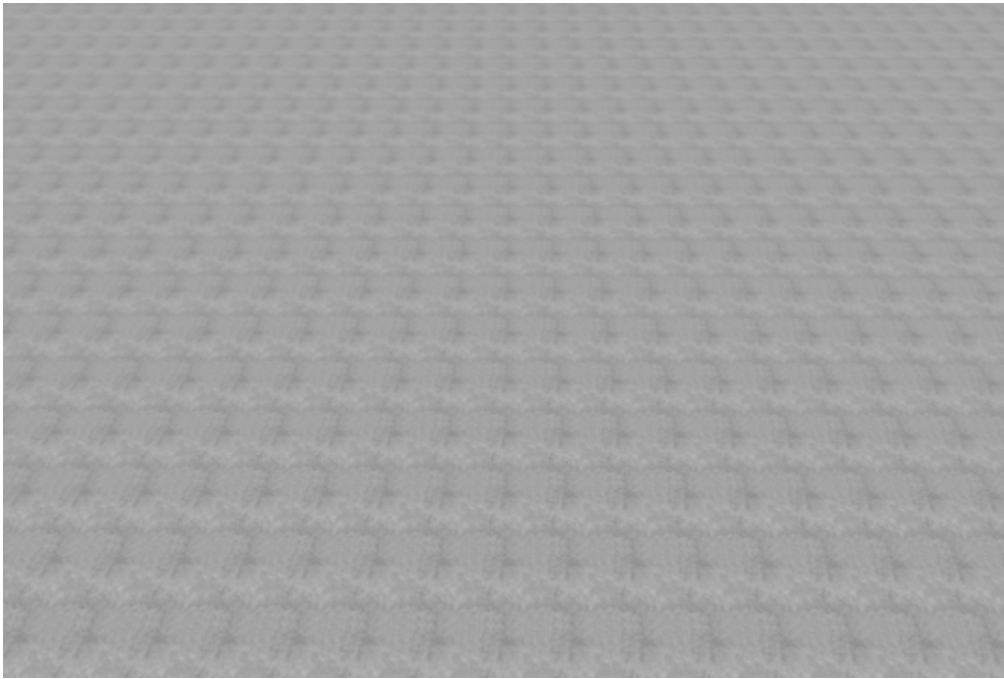
Mosaic Tiling

The idea of patterns is a common one for the human eye, we are very good at detecting visual patterns and we can cognitively recognise when we see one. So this becomes an important problem to solve when we want to make a surface like the ground look pseudo random, meaning enough randomness for us not to notice its repeating pattern underneath the noise we add.

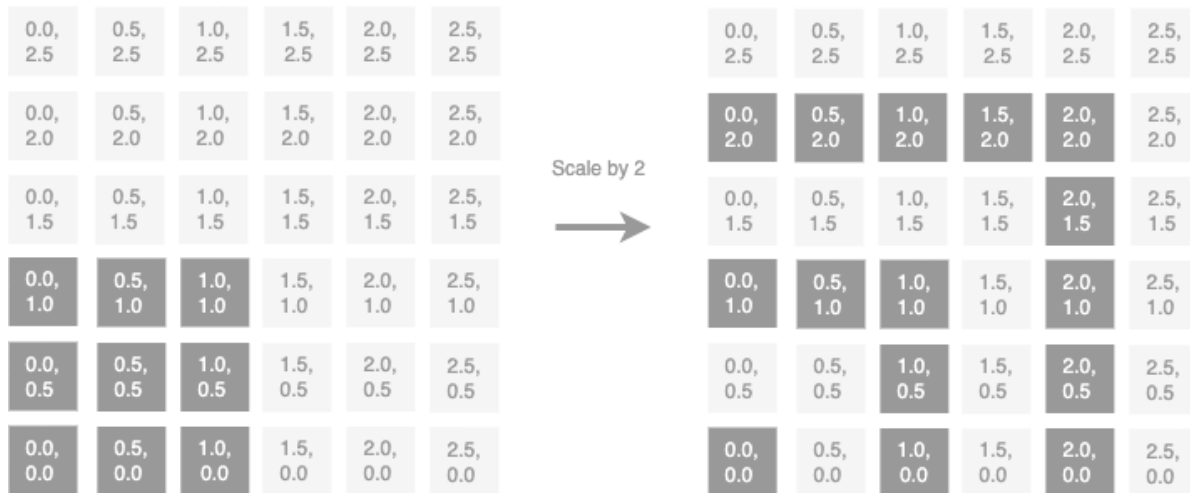
A side note is we often have signals in nature and in computer graphics and there will be noise on top making it hard to discern what something is made of. In nature this happens all the time, like light from the sun will vary in position over time, it will hardly if at all ever follow the exact same path again and again. What I mean by this is we have the same foundational building blocks like one texture and then add noise to rotate the texture 100's of times so the ground looks noisy or more random.

Why is this useful? So if we have semi seamless textures we can avoid the repeating patterns you would inevitably see and make the plane or ground look more natural like what we find in nature. Which is often the aim in gaming graphics.

Ok so we have a texture; If we just repeat this on a plane with this setup the texture would have this repeating pattern, This is obviously not good for anyone. To modify this we need to randomly rotate the UV's:



We need to scale the UVs. The example below is very generalistic but shows my point visually. If we multiply the top red square, [1.0, 1.0] by an arbitrary 2.5. We are essentially saying we want the data from the UV position outside of our current space which is [2.5, 2.5] and we want to set this to the position [1.0, 1.0] for this new data.

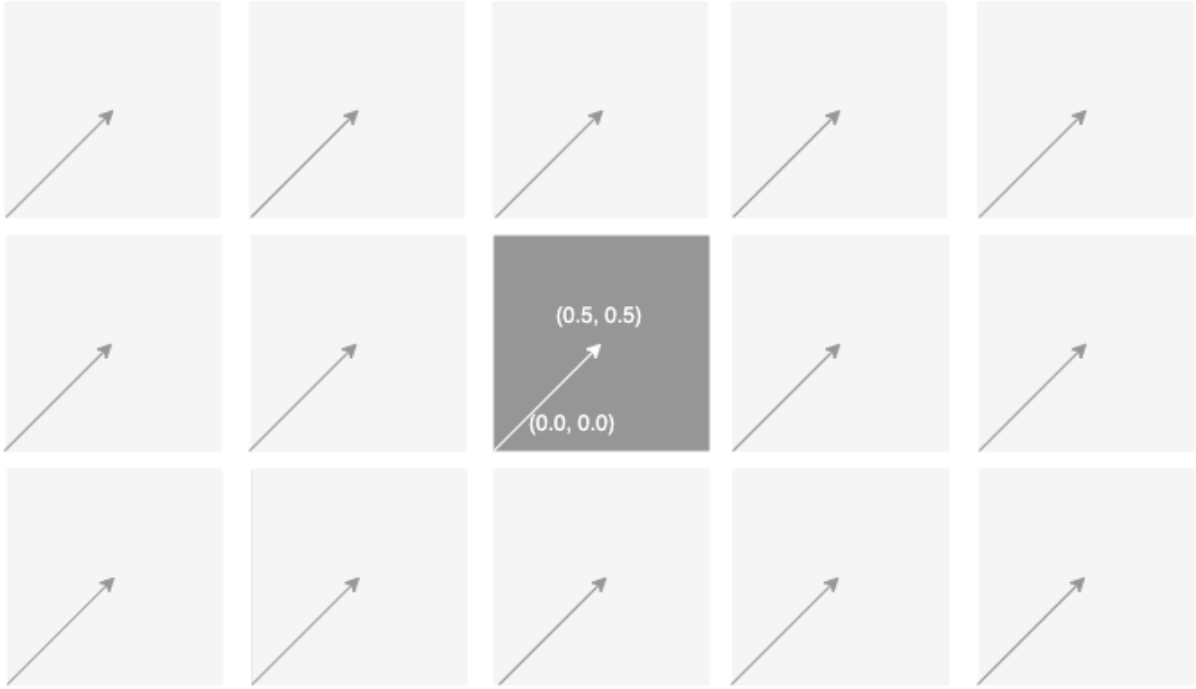


So we sort of squeeze in more data by modifying the UVs in a scaled way either by multiplying and zooming out or dividing and zooming in. We need to ensure each tile of repeating texture has the same center. When we floor the modified uv we could have [5.5, 5.2] and the floored value becomes [5.0, 5.0] and any value in the range [5-6, 5-6] becomes [5, 5]. Why is this important? Well we need to ensure each fragment in the range 5-6 has the same center of rotation as stated above.

```
vec2 tile = floor(uv);
```

Given we now have the same center of rotation we also want to center the rotation so for each interval or tile, as currently we have the bottom left furthest point, so we do this:

```
vec2 center = tile + vec2(0.5, 0.5);
```



We now want to rotate the UVs around a center point with a random angle of rotation.

```
vec2 randomRotatedTileUV = rotateUV(  
    uv,  
    rand(tile, 2.0) * 10.0,  
    Center  
);
```

The random function can be found by googling stackoverflow answers and can be done like this. Most functions you will come across are pseudo random, meaning they aren't truly random but follow patterns which mimic randomness. Hash functions are used a lot in this area, they provide pseudo randomness.

```

float rand(vec2 co, float seed) {
    float a = 12.9898;
    float b = 78.233;
    float c = 43758.5453;
    float dt= dot(co.xy ,vec2(a,b));
    float sn= mod(dt, 3.14);
    return fract(sin(sn + seed) * c);
}

```

The rotation function accepts a uv coordinate, a rotation and a mid point for which the UVs will be rotated around.

```

vec2 rotateUV(vec2 uv, float rotation, vec2 mid) {
    return vec2(
        cos(rotation) * (uv.x - mid.x) + sin(rotation) * (uv.y - mid.y) + mid.x,
        cos(rotation) * (uv.y - mid.y) - sin(rotation) * (uv.x - mid.x) + mid.y
    );
}

```

After all this we then sample the repeated texture like so:

```

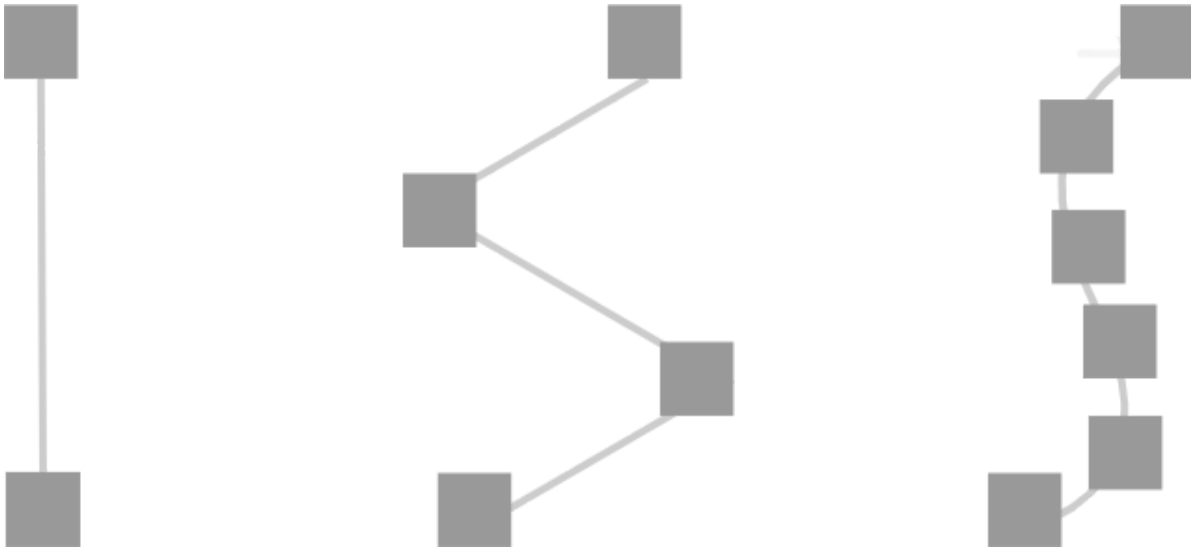
gl_FragColor = texture(tDiffuse, randomRotatedTileUV);

```

The full code example can be found in the appendix under 1.2. This is a really good technique to cover a large area with a texture and minimise the repeating pattern effect. Which human eyes are very good at noticing.

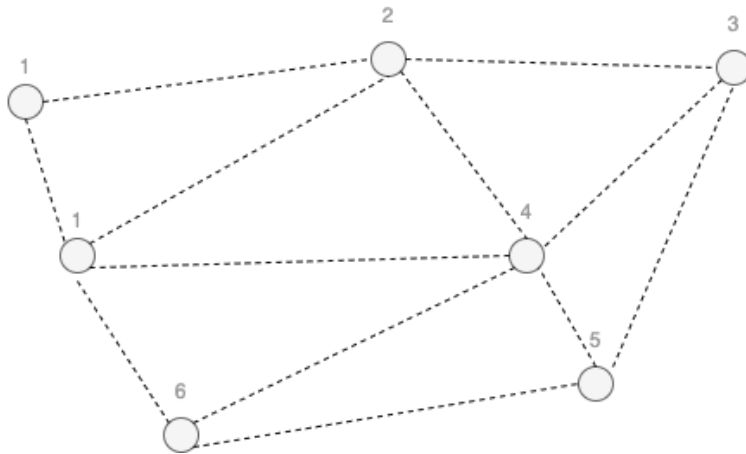
Subdivision modifier

Why would we want to divide a mesh into more vertices? To answer this let's take a look at a plane side on with noise:



As we increase the number of vertices we allow for more detailed meshes, more points in 3d space can be defined, allowing for more curvature, demonstrated above.

The purpose is now clear, so how can we accomplish this? Let's take another look at geometry and in particular `bufferGeometries` in `three.js`. `BufferGeometries` store the vertex data along with all other per vertex attributes like indices or UV's or normals.



In the above example we would have this for the indices of the buffer geometry. The indices are what determined the edges to each triangle and ultimately make up the faces of our geometry or surface.:

```
// Example of an indexed bufferGeometry
vertices = Float32Array([
  vertexOne.x, vertexOne.y, vertexOne.z,
  vertexTwo.x, vertexTwo.y, vertexTwo.z,
  vertexThree.x, vertexThree.y, vertexThree.z,
  vertexFour.x, vertexFour.y, vertexFour.z,
  vertexFive.x, vertexFive.y, vertexFive.z,
  vertexSix.x, vertexSix.y, vertexSix.z,
  vertexSeven.x, vertexSeven.y, vertexSeven.z
]);

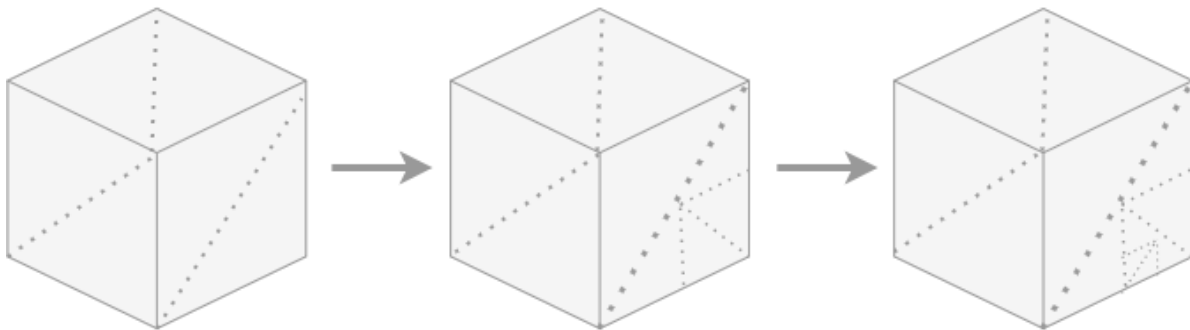
Indices = int8Array([
  1, 2, 4,
  1, 7, 4,
  7, 4, 5,
  7, 6, 5,
  2, 4, 3,
  4, 3, 5
]);
```

This works as the vertices are in the sequential order, the index relates to the position of the vertex in the vertices array. If we put vertexOne at the end, then its index would be the last one of the array of 3's. And the indices array would have to change

accordingly. A key point to remember is that every mesh in three.js is by default made of triangles. And this shows up in the indices. We store the index of each vertex for the triangle. When we have all the info stored, the shader pipeline will construct triangle faces based on this information.

Every vertex we store 3 values in the `float32Array`, the x/y/z values. And we should note that we always used typed arrays for these data attributes, as it is a lot more efficient than using a normal array. This is standard practice when dealing with large amounts of data and vertex data gets large very quickly.

Let's examine what we want to accomplish here:



The idea is simple, we use a pointer event on a three.js mesh which is powered behind the scenes by a raycaster (a side note you should use these sparingly as they are expensive in terms of computing resources).

This gives us the face via the indices. We can then calculate the new vertices we would need and finally construct the faces and indices we would need. Now we know how the indices work and therefore know how to construct faces.

```
// i.e. we get the faces via this event object produced from a pointer
event
const aIndex = e.intersections[0].face.a;
const bIndex = e.intersections[0].face.b;
const cIndex = e.intersections[0].face.c;
```

This makes perfect sense with what we said previously about each face storing the index of each vertex and as meshes are made of triangles we have the above a/b/c data points for the face.

Ok, so we need to generate 3 new indices. We CANNOT just reuse the existing vertex indices. This is because we would just construct new faces between the vertices of what we already have, this is not what we want. So we just add 1, 2 and 3 onto the length of total number of vertices, like so:

```
const newAIndex = numVertices;
const newBIndex = numVertices + 1;
const newCIndex = numVertices + 2;
```

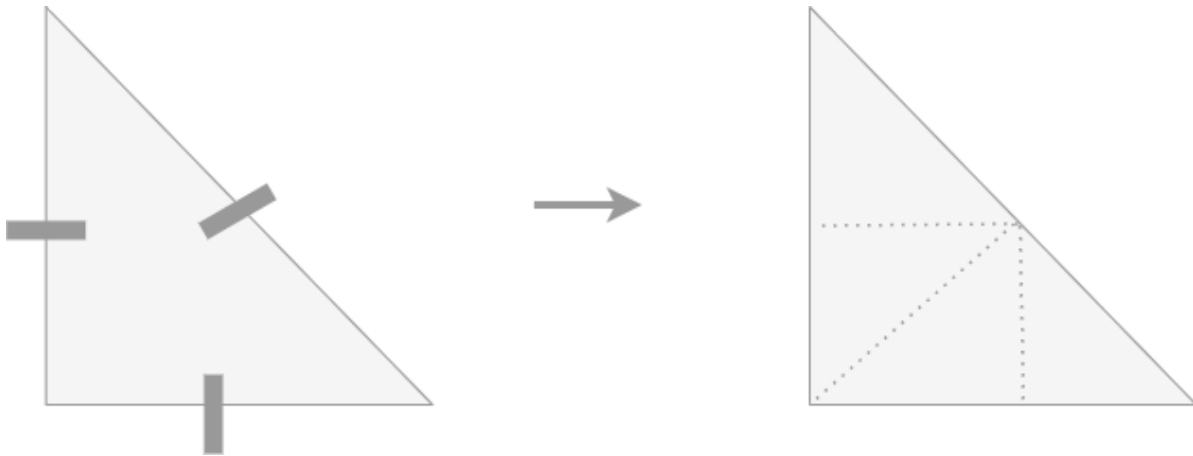
Remembering that indices are zero origin based. Meaning the first index will be stored at position 0, quite a repeating pattern in most western programming languages.

The positions of each of the selected face can be acquired like so in three.js:

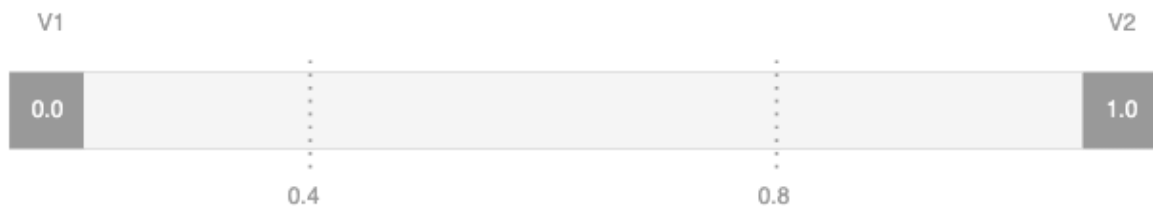
```
// This is grabbing the position using the intersections face indices
const aPosition = new THREE.Vector3().fromBufferAttribute(position, aIndex);
const bPosition = new THREE.Vector3().fromBufferAttribute(position, bIndex);
const cPosition = new THREE.Vector3().fromBufferAttribute(position, cIndex);
```

This is a way to do a lookup in the position attribute, we are saying at index aIndex,

what position is it? And by the way, store it in the Vector3. So there is nothing fancy here, all we are going to do is get the midpoint of each pair of vertices in the selected triangle face.



A useful mathematical function to do this is lerp. Lerp is producing a new vector between two vectors based on a percentage of A or B, i.e.



```
vec3 = lerpVector(vec3(1.0, 0.0, 0.0), vec3(0.0, 1.0, 0.0), 0.4);  
vec3 = lerpVector(vec3(1.0, 0.0, 0.0), vec3(0.0, 1.0, 0.0), 0.8)
```

We are effectively saying what is 40% the way from A to B? or what is 80% between A and B? One real world example would be to do this in THREE.js:

```
// we can then find the midpoint between each corner  
const abPosition = new THREE.Vector3().lerpVectors(  

```

```

    aPosition,
    bPosition,
    0.5
  );

  const bcPosition = new THREE.Vector3().lerpVectors(
    bPosition,
    cPosition,
    0.5
  );

  const caPosition = new THREE.Vector3().lerpVectors(
    cPosition,
    aPosition,
    0.5
  );

```

This gets the midpoint between the existing vertices and now we have the new vertex positions.

```

// Just remember 3 indices = 1 face drawn
newIndices.push(
  aIndex,
  newAIndex,
  newCIndex,
  newAIndex,
  bIndex,
  newBIndex,
  newAIndex,
  newBIndex,
  newCIndex,
  newBIndex,
  cIndex,
  newCIndex
);

// we only want to add new vertices and not the old ones otherwise we get
// duplication. In fact we want to remove the old vertex.
newVertices.push(
  abPosition.x,
  abPosition.y,
  abPosition.z,

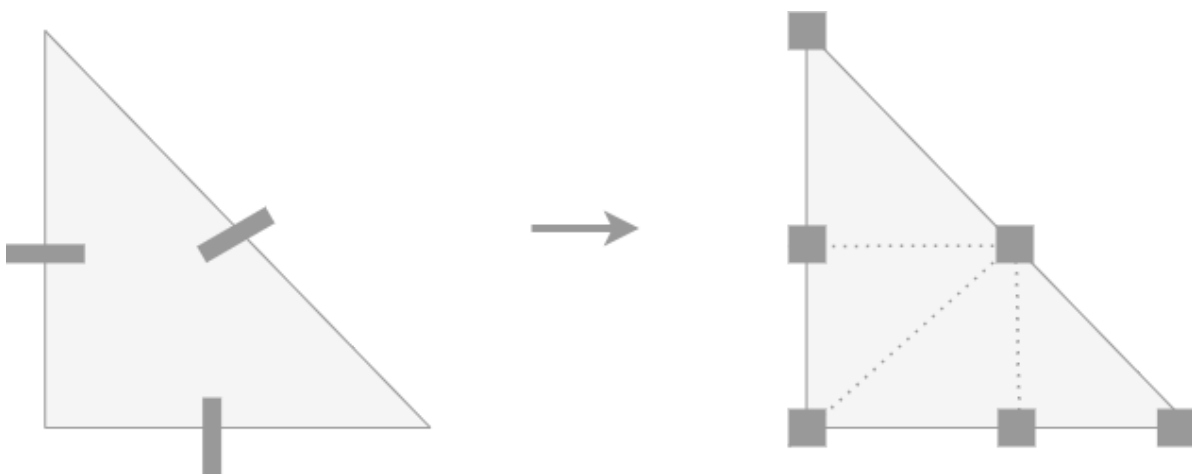
```

```

    bcPosition.x,
    bcPosition.y,
    bcPosition.z,
    caPosition.x,
    caPosition.y,
    caPosition.z
  );

```

This diagram below walks you through what is going on in the above code. Going from one face and 3 vertices to 6 vertices and 4 faces. With each vertex having 3 components and each face having 3 indices and some of these indices will be used more than one, which is completely fine and how indices work with bufferGeometries.



For the keen eyed among you, you might realize we still have the original face in the indices array, don't we need to remove this? And yes, yes we do. To filter out the 3 indices of the original red face, we use a function like this:

```

function filterArray(source, arr) {
  const result = [];
  for (let i = 0; i < arr.length; i += 3) {
    const index1 = arr[i];
    const index2 = arr[i + 1];

```

```

    const index3 = arr[i + 2];

    if (
      source.includes(index1) &&
      source.includes(index2) &&
      source.includes(index3)
    ) {
      continue;
    }

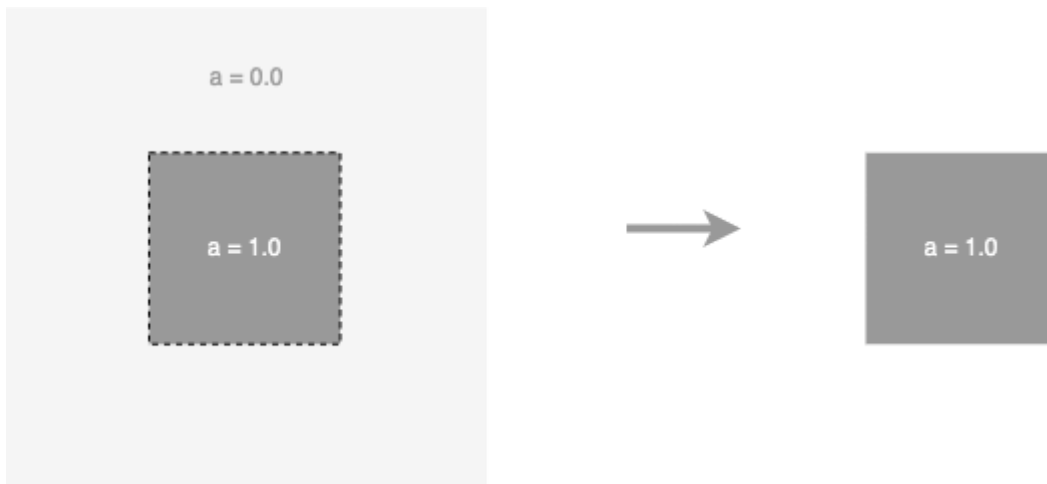
    result.push(index1, index2, index3);
  }
  return result;
}

```

I'm not going to go into the details of the rest of the mini project, but if your interested, a full code example can be found in the appendix (1.3).

Animated shader - alpha clipped grass

Alpha clipping is where anything below a threshold for alpha is discarded. If we discard something in a fragment shader, nothing is rendered for the fragment. This is particularly useful with materials and shaders in three.js. Below would discard anything which is not red (the purple represents areas of the texture which have an opacity below the threshold):



Alpha just like any other channel or component ranges from 0.0 - 1.0 and we must define transparency on the material class definition itself as well or we will not see any opacity on the material. Don't forget materials should have the double sided setting set for each side to be seen.

So what would an if else look like with the discard key word in GLSL:

```
//If transparent, don't draw  
if (alpha < 0.005) discard;
```

So how do we get the wavy effect on the grass? First of all we need something which changes over time on repeat. Sin and Cos functions are perfect for repeating waves. As when you go above the 2π angle for cos and sin it just repeats the pattern with a different magnitude.

We apply sin or cos to a direction or axis by using that component of the vector. So if we want to apply sin and imprint the sin pattern on the y axis we use `myVector.y` as

one of the inputs for cos. function. i.e.

```
float wavyStrength = sin(time + pos.y)* 0.04;
```

We can change the amplitude of strength of the sin wave by only getting 4% as we do above. An important note the position or pos varying is passed from the vertex shader so although the plane we use only has 4 vertices, they will be interpolated or a gradient created as previously described in the fragment shader runs. Now we want to move the texture in the x axis, firstly we want the waviness to be different the further up the y axis we go so we do this and apply a sine wave to the y axis:

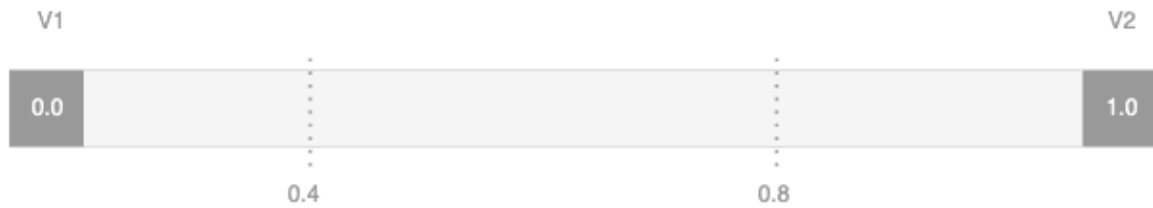
```
float wavyStrength = sin(time + pos.y)* 0.04;
```

So we can do a mix of the x component and some noise value with our modulator of the y axis. This way we can apply a different noise value for every vUv.x and have the noise value oscillate back and forth on the x axis.

```
vec2 uvs = vec2(mix(vUv.x, noise * scaledNoise , wavyStrength ), vUv.y);
```



Mixing is much the same as lerp'ing, remember this:



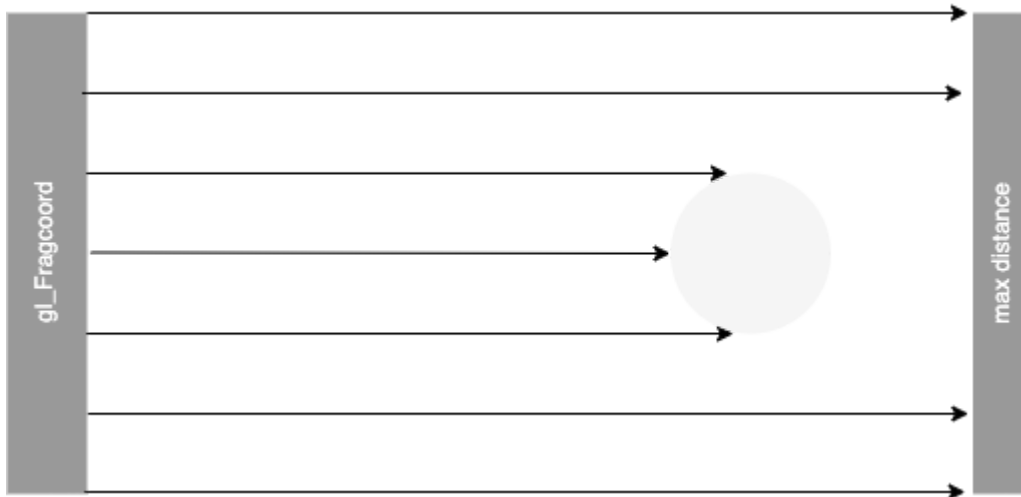
```
vec2 uvs = vec2(mix(vUv.x, noise * scaledNoise , wavyStrength ), vUv.y);
```

Mixing above is like adding smooth noise to each position on the axis at each y axis level. Below is the result and the full code example can be found in the appendix (1.4).



Ray marching over a Three.js scene

This is an interesting concept. Ray marching has been described previously whereby you increment rays from every fragment or pixel and use SDFs as a way to calculate if the ray is inside or outside of the defined shape equation (in a purely math world). We can then dictate what color to set that pixel. But how does this all fit in with the original scene?



If we just raycast and don't consider the coordinate system matrices, when we move the camera we will be in different coordinate systems and it won't work. The color of the current fragment won't take into account the various different coordinate systems and won't be in the right place giving odd results on the screen as a whole. If we were to have a stationary camera and you couldn't move it then it would work to a certain degree.

Let's take a deeper look at the coordinate systems:

```

vec3 ray = (
    modelMatrix * cameraProjectionMatrixInverse * vec4( clipSpace.xy, 1.0, 1.0 )
).xyz;

```

This code allows us to go from clipSpace ranging from [-1, 1] which is what this calculation does

```

vec2 uv = ( gl_FragCoord.xy * 2.0 - (uResolution - vec2(0.5,0.5)) ) /
uResolution;
vec2 p = gl_FragCoord.xy / uResolution.xy;

vec2 clipSpace = uv;

```

To view space [-inf, inf], view space remember is just the view from the camera. So we have the view from the camera but this isn't enough.

Think of it like this, when the camera moves, everything rotates in the scene. So if we don't apply rotation and other transforms to the ray, the ray will not rotate correctly with respect to the scene and as the camera rotates, the sphere SDF won't stay in the correct place, it will move as we move the camera. Which is obviously not what we want.

The ray marching algorithm has 3 pieces to it; it has the origin, the ray and the distance traveled.

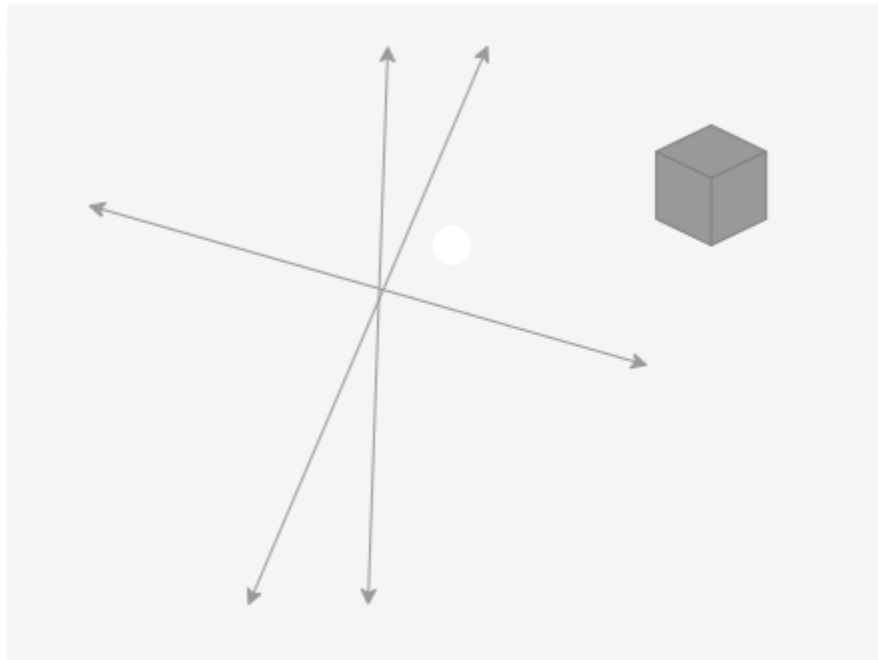
```

// Ray Marching Variables
vec3 rayOrigin = cameraPosition;
vec3 rayDirection = ray;
float rayDistance = 0.0;

```

The origin is the camera's position, the ray is what gives us direction and the magnitude comes from rayDistance, which incrementally increases as the loop

iterations increase. The magnitude of the incrementation determines the resolution or the amount of detail we see with the SDF border, if we have very large increments we might jump into the SDF quite far so it won't portray the border of the SDF very accurately. Always something to bear in mind. Below shows a sphere raymarched onto a three.js scene.



The last bit is where we do

```
vec3 sum = texture(tDiffuse, p).xyz;
float rayDistance = 0.0;
float MAX_DISTANCE = 2000.0;

for (int i = 0; i < 1000; i++) {
    vec3 currentStep = rayOrigin + rayDirection * rayDistance ;
```

```

float dist = sphereSDF(currentStep, sphereOirgin, .10);

if (dist < 0.00001 ) {
    sum = vec3(1.0);
}
if (rayDistance > MAX_DISTANCE) {
    break;
}
rayDistance += 0.09;
}

gl_FragColor = vec4(sum.xyz, 1.0);

```

A few points to emphasis here. Firstly tDiffuse is a common conventional name and is a texture of the entire scene, particularly in post processing. If you look for the components of postprocessing in the three.js repo you will likely stumble across the name tDiffuse. 'T' stands for texture and diffuse often is analogous to color, so we are saying this is a snapshot of the canvas screen which contains all the color information. You'll also probably spot this on shader toys as well.

The convention comes from the practice of taking the first letter of the type like vec2/3/4 or texture or int or float:

```

// Variable naming examples in GLSL
vColor === vector + color
iColor === int + color
tColor tDiffuse === texture + color

```

You always define a max distance for the ray incrementation as this is a fail safe, it will cut off the loop if we go past the max distance and still haven't hit inside a SDF shape function. And you use the keyword break to exit the for loop, in this case the for loop is the mechanism by which we do the incrementation of the ray.

We do this in every fragment or pixel and every frame because in every frame we want to check if this fragment or pixel is inside the SDF shape function.

If you come across layered noise in this book you will understand that for loops should be carefully considered, as they can be very intensive when you have a loop within a loop within a loop. Which commonly occurs in volumetrics when you are creating octaves of noise.

In the code below we get the signed distance and check if this pixel or fragment is inside the sphere. If it is inside the sphere we set the sum color to be white. At the end of the for loop code, we increment the rayDistance and continue the loop. This propagates and grows the ray into the scene.

This sum will be white if inside and will show the normal screen texture if not inside the sphere. This means we can modify the original screen's texture, which will contain all the correct colors and positions of all the scenes materials and geometries, while amalgamating the screen texture with the artificial SDF colors by replacing the screen texture pixels with white in the appropriate places, which is dictated by the SDF function.

Below shows the algorithm in action in a simple three.js scene:

```
uniform vec3 camPos;  
uniform vec2 uResolution;  
uniform sampler2D tDiffuse;  
uniform mat4 cameraWorldMatrix;  
uniform mat4 cameraProjectionMatrixInverse;  
varying vec2 vUv;
```

```

float sphereSDF (vec3 position, vec3 origin, float radius) {
    return length(origin-position) - radius;
}

void main() {
    vec2 uv =( gl_FragCoord.xy * 2.0 - (uResolution - vec2(0.5,0.5)) ) /
uResolution;
    vec2 p = gl_FragCoord.xy / uResolution.xy;

    vec2 clipSpace = uv;

    vec3 ray = (modelMatrix * cameraProjectionMatrixInverse * vec4( clipSpace.xy,
1.0, 1.0 )).xyz;
    ray = normalize( ray );

    vec3 cameraPosition = camPos;

    // Ray Marching Variables
    vec3 rayOrigin = cameraPosition;
    vec3 rayDirection = ray;
    vec3 sphereOrigin = vec3(0.30);

    vec3 sum = texture(tDiffuse, p).xyz;
    float rayDistance = 0.0;
    float MAX_DISTANCE = 2000.0;

    for (int i = 0; i< 1000;i ++){
        vec3 currentStep = rayOrigin + rayDirection * rayDistance ;

        float dist = sphereSDF(currentStep,sphereOrigin, .10);

        if (dist < 0.00001 ) {
            sum = vec3(1.0);
        }
        if (rayDistance > MAX_DISTANCE) {
            break;
        }
        rayDistance += 0.09;
    }

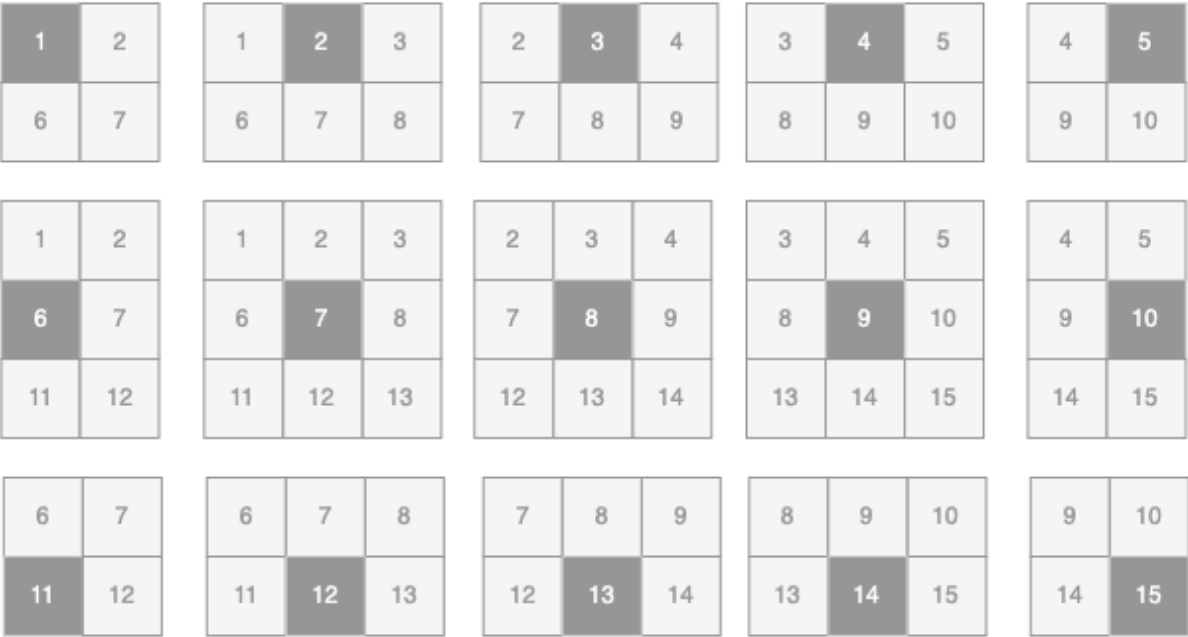
    gl_FragColor = vec4(sum.xyz, 1.0);
}

```

Volumetric lights and radial blur

What is blurring in terms of GLSL, lets go really basic and visualize the process with integers and floats. The idea is we average the current square with its surroundings like so:

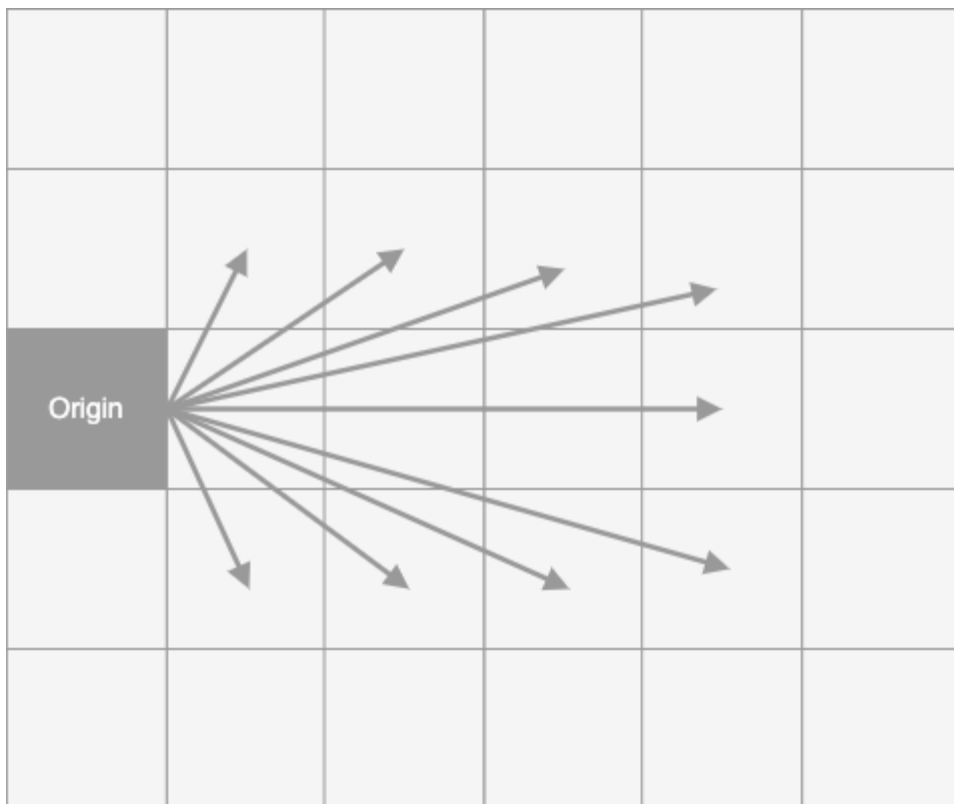
1	2	3	4	5
6	7	8	9	10
11	12	13	14	15



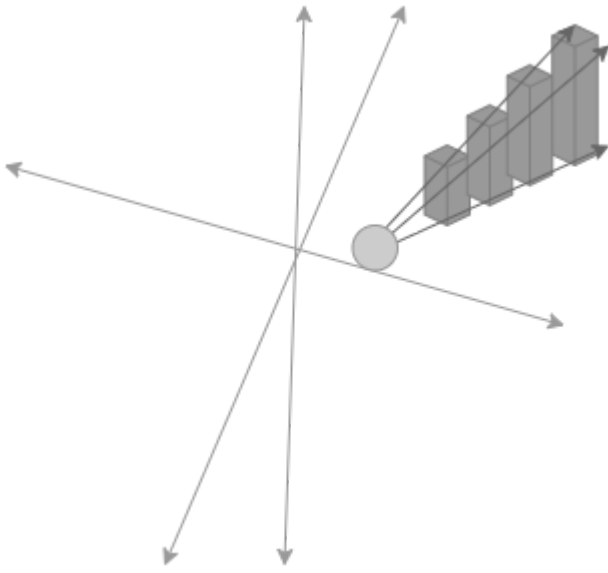
4	4.5	5.5	6.5	7
6.5	6.44	8	9	9.5
9	9.5	10.5	11.5	12

The idea is that we homogenize the data in both the horizontal and vertical directions. In essence this is basic blur. The range of our data decreases and we homogenize, and that is it.. That's how you blur in general and in GLSL, you homogenize or average the surrounding data points of the current data point.

To understand what we are trying to do with radial blur, we have to understand what radial blur is. From our favorite chat bot: "Radial blur is a visual effect that creates a blurring motion radiating outward or inward from a central point". We can think of radial blurring as smudging from an origin. We average along a vector.



We know postprocessing is what we are going to want to use as it affects the entire scene. It's no good to have the effect confined to the surface of a mesh. We need it to be around the mesh. Below is a more accurate example of radial blur:



The thing we have control over is how and where we sample the tDiffuse or screen texture with all the scenes color information in. So we can do the radial blur on the sampling of this texture. So we have to decide how many samples and in what direction?

```
const int NUM_SAMPLES = 80;

vec3 light_on_screen = vec3(0.5, 0.5, 0.0);

// *****

vec2 deltaTextCoord = vec2(vUv.xy - light_on_screen.xy);
vec2 textCoo = vUv.st;
deltaTextCoord *= 1.0 / float(NUM_SAMPLES) * density;
```

The direction is from the middle of the UVs which is the middle of the screen and by default the middle of the screen is where the origin is `vec3(0.0, 0.0, 0.0)`.

In thinking of directions and vectors, if we want to go from point A to B it is $B - A$. So `light_on_screen` to `vUv` means `vUv - light_on_screen`.

The next bit is to determine how often we are going to sample along the direction or vector. To do this we do the below pseudo code:

```
Distance from light / Total number of samples * modulator for density
or:

frequencyOfSamples === deltaTextCoord *= 1.0 / float(NUM_SAMPLES) *
density;
```

The next thing is to average along the vector or direction `deltaTextCoord`.

The for loop is used to average with the number of samples being the total number of iterations:

```
for(int i=0; i < NUM_SAMPLES ; i++) {
    textCoo -= deltaTextCoord;

    textCoo.s = clamp(textCoo.s, 0.0, 1.0);
    textCoo.t = clamp(textCoo.t, 0.0, 1.0);

    vec4 sample1 = texture2D(tDiffuse, textCoo);

    sample1 *= illuminationDecay * weight;

    c += sample1;

    illuminationDecay *= decay;
}
```

Let's unpackage this. The `textCoo` are the UVs, the amount and direction we will sample are stored in `deltaTextCoord` from the calculations we made above. So now we are sampling by a fixed amount each iteration of the for loop in a certain direction. We clamp the UVs to [0.0, 1.0]. Then we do the texture lookup or sampling via:

```
vec4 sample1 = texture2D(tDiffuse, textCoo);
```

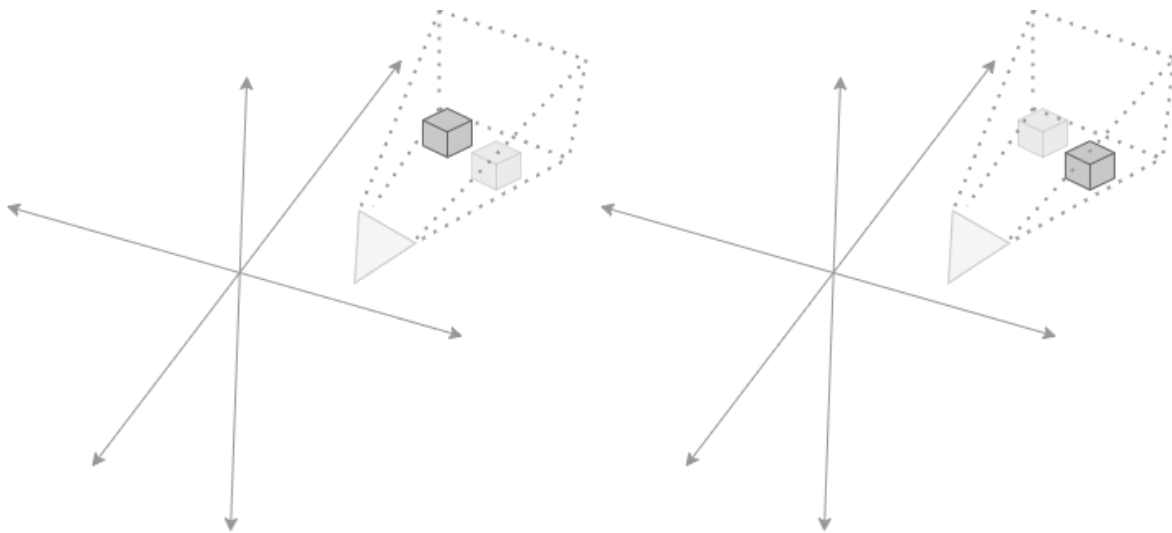
And the trick is we want the radial blur to be less bright or have less data the further away you are from the origin. And as we sample in a direction and the higher the for loop iterations the further away we are. We want to decrease the strength of the colors so we use the decay pattern:

```
sample1 *= illuminationDecay * weight;  
  
illuminationDecay *= decay;
```

Sample1 is the color we got via the sample, we decrease the amount of color or brightness or data by multiplying by less than 1.0. And as the for loop progresses we decrease the decay and by proxy decrease the color of the radial blur the further you are away from the radial blur center.

We have just gone through a basic example but how can we apply this radial blur to an object in a three.js scene? We need layers. What are layers?

Layers are rendered in a way which isolate meshes while rendering whats in each layer separately, like so:



Layers in three.js work by setting a layer on the mesh like so and rendering the layers in useFrame using the camera settings.

```
<mesh name="mesh1" layer={1} />
```

```
<mesh name="mesh2" layer={2} />
```

```
// and this is how we set and render the layers
useFrame(({ gl }) => {
  camera.layers.set(2);
  gl.render(scene, camera);
  camera.layers.set(1);
  gl.render(scene, camera);
});
```

We can access the webGL context which is often just shortened to gl like we have above. This is frequently seen in pure openGL as the main focal point for everything around 3D scenes in openGL. In three.js we primarily access it via useFrame, which runs every frame. We need to therefore be mindful not to call any intensive functions inside the useFrame, obviously its project specific what fps the app should be but it depends what the requirements are. Usually we aim for 60fps as standard on the low end devices. It is useful to state that you should not use state in the useFrame, but you can always use refs and pass the refs between components. Like so:

```
// Component 1
useFrame(() => {
  ref.value = true;
}, 1)
```

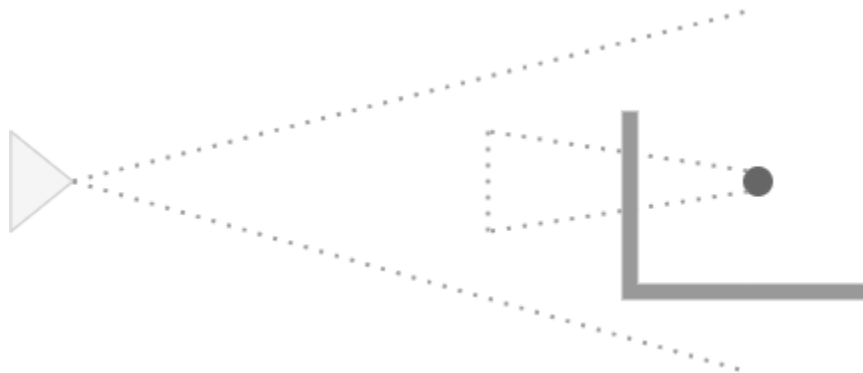
```
// Component 2
useFrame(() => {
  if (ref.value) {
    // do something
  }
}, 2)
```

This is not an official pattern but it can be used and is a completely valid way to bypass state in React Three Fiber between useFrames. Although team standards might differ

team to team, this is a neat solution which does work between components frame to frame (in some circumstances). You can find the full code for this in the appendix (1.5). You'll notice this is very similar to the basic radial blur. Except we use the depth to world coordinates function again and then do this if else:

```
if (length(worldPos) > length(camPos) || camPos.z >= worldPos.z) {  
    gl_FragColor = vec4(c.rgb);  
} else {  
    gl_FragColor = textureSample.rgb;  
}
```

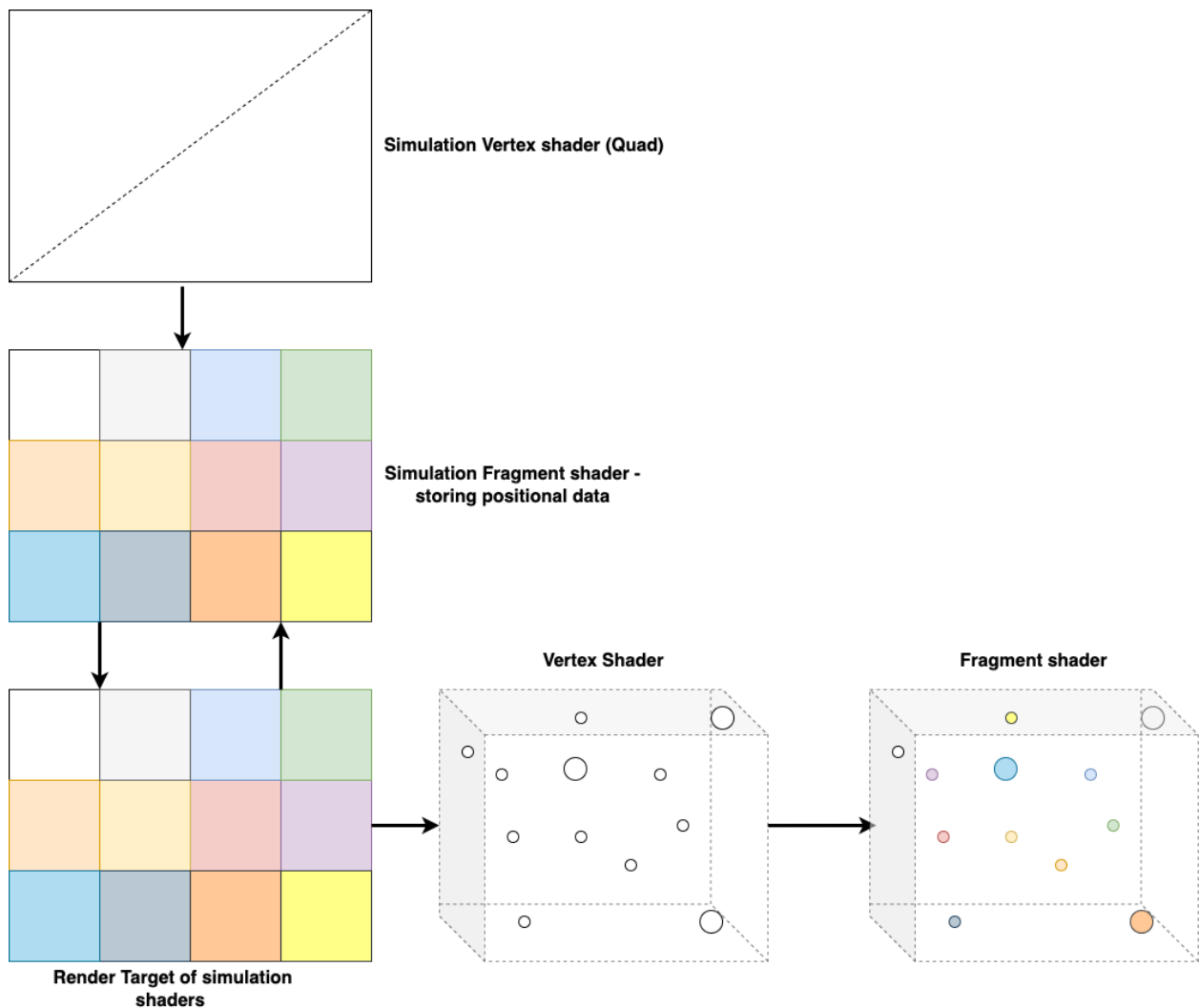
This is saying in layman's terms, is the camera further away than the object in the scene and if it is then we want to show the radial blur - are we in front of the object. This works, remember, as length in GLSL gives positive and negative values. When the camera's position goes into the negative it won't be greater than the object to the origin. And in this case show the original objects color/texture.



This is somewhat fixed to the way the scene is rotated and positioned, but shows what's possible with some thought about the scene and tools at your disposal. There's always ways around things. The full code can be found in appendix 1.6.

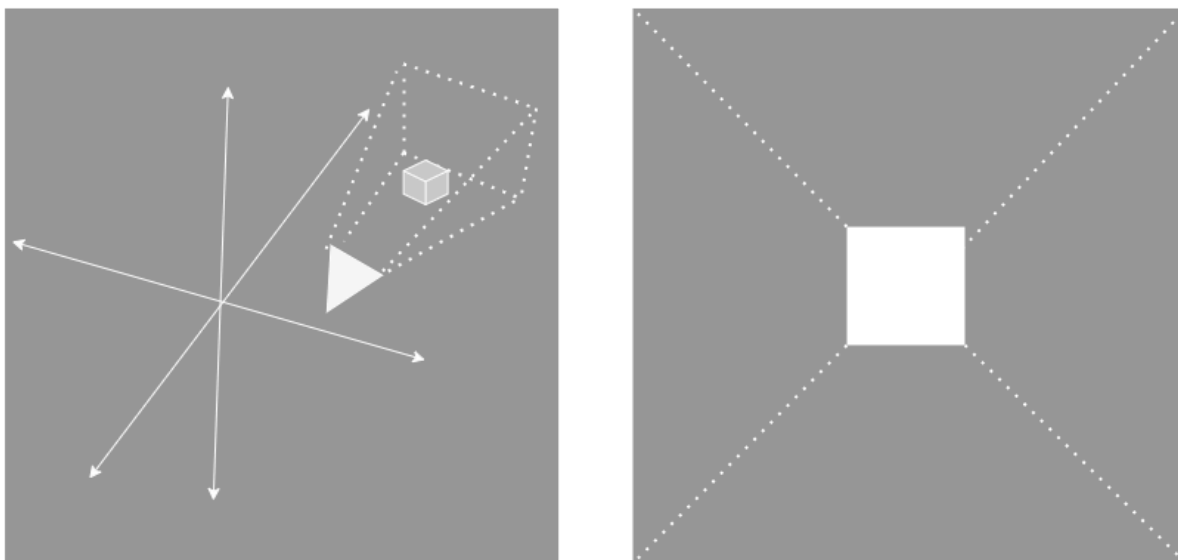
GLSL and GPU Particles

The essence of GPU particles involved two concepts you might not have been fully aware of. Which are texture feedback and off screen renders i.e. a simulation shader and a render shader. GPU particles are good as it avoids the bottleneck which occurs when using more traditional CPU particles. Mainly because CPU particles require a lot of sequential data setup and manipulation whereas we can leverage the GPU to do a lot of the calculations in parallel, making them A LOT more efficient.



Consider the above diagram. Firstly we have a quad which is a simple pair of triangles (or a single triangle sometimes) which fills the entire screen, A screen quad is useful because it gives us vertices so we can run the vertex shader, but minimalistically. Then this gives us access to the fragment shader which can have 100's of runs dictated to by the size of the screen or quad. Remembering that for every pixel we will have a fragment shader run, so although we have 4 vertices we can have thousands of fragment shader runs, much more efficient than a scene full of rich and detailed

What do I mean by an offscreen render?



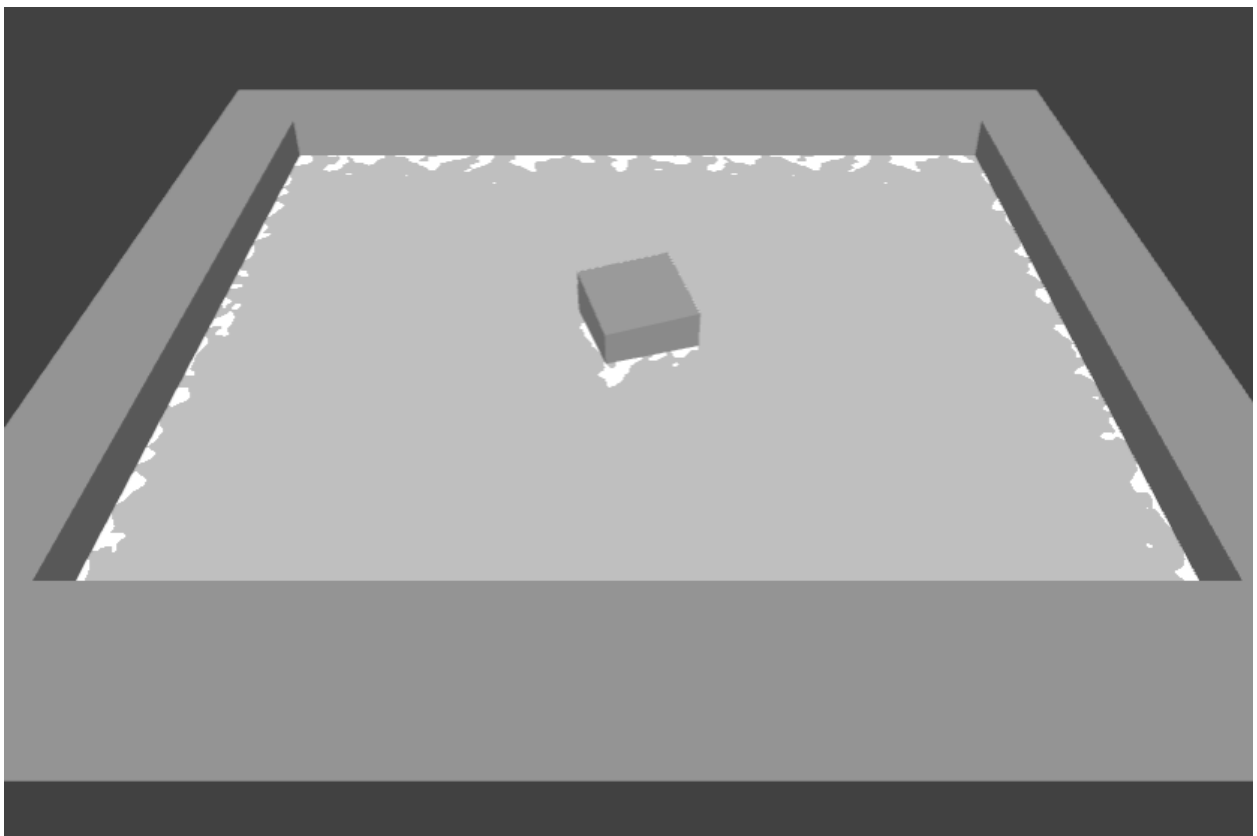
In our scenario the left is the onscreen camera which renders the particles and the right is the orthographic simulation shaders. The offscreen render pattern usually follows the idea of a 1x1 plane and then the orthographic camera is positioned in such a way as to look towards and cover the entire plane perfectly. In essence an offscreen

quad which gives us access to many more fragment shader runs as stated above. The orthographic camera is used so we don't have to calculate or worry about perspective which can make calculations and position of the camera very easy in this scenario.

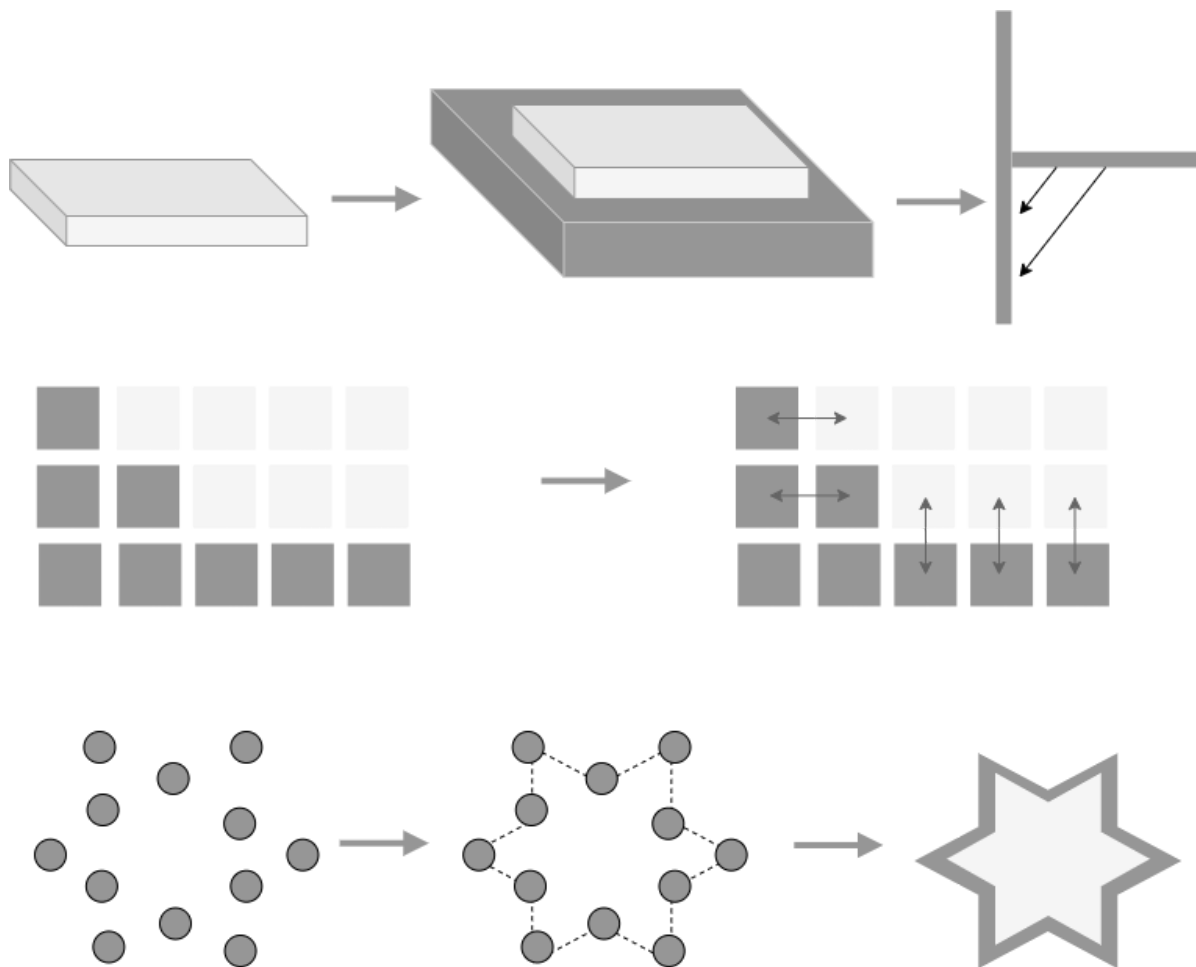
(FINISH WHEN CODE IS UPDATED)

Edge detection and concave hulls

Edge detection is intertwined in many 3D effects. To name one among many is water and foam. The way we add foam to a body of water is that we detect the edge and then use this edge in a fragment shader to position foam like so:

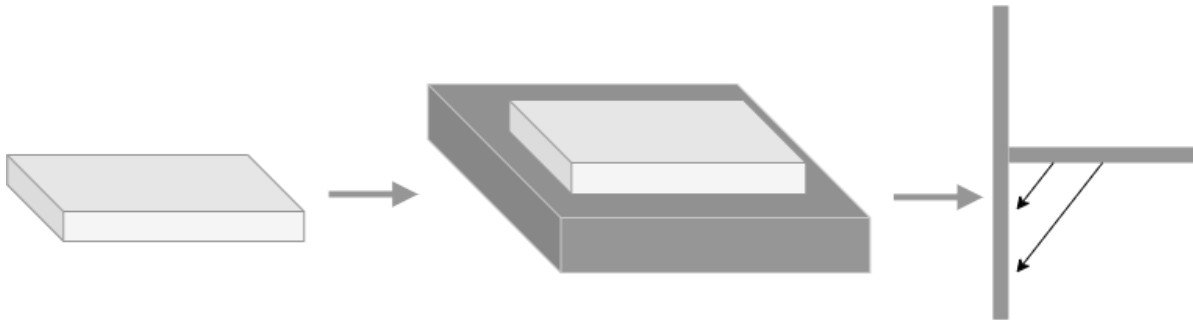


The diagram below illustrates the 2 common ways and the new method proposed here. The first is depth textures, the second is partial derivatives and the new method is a novel edge detection using vertex data:



The three methods above are unique in their approaches to edge detection. The first method uses depth, the second uses partial derivatives and the third uses vertex data of a 2D polygon shape. Let's go through the first method.

The first method uses postprocessing and depth textures, the first diagram shows the comparison we use to detect edges:



(check this.....)

```
vec3 sideDepth = texture(textureSideDepth, vUv);  
vec3 groundDepth = texture(textureGroundDepth, vUv);  
  
// Remembering distance gives absolute values of distance between point  
// A and B  
float distanceOfDepth = distance(sideDepth, groundDepth);  
  
float depthThreshold = 0.6;  
  
// Shorter arrow  
distanceOfDepth = 0.5;  
// = foam  
  
// Longer Arrow  
distanceOfDepth = 1.1;  
// = no foam
```

The idea here is we render the 2 components (ground and side). Then we sample the side depth and also sample using the same uv's on the depth of the ground. After which we get the distance between the two values, this will then allow us to say if the distance is small we can add foam and if it is a long distance away we don't add foam or the effect.

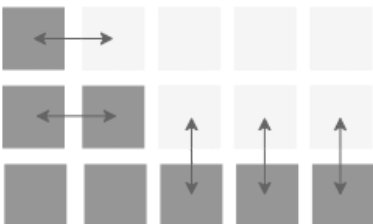


The second method uses partial derivatives which we went through earlier in the book. A quick reminder that partial derivatives allow you to sample the right or upper pixel or fragments color or data value. In this case it can be incredibly useful. Why?

```
vec3 white = vec3(1.0, 1.0, 1.0);
vec3 grey = vec3(0.2, 0.2, 0.2);

float dist = distance(white, grey);
// = vec3(0.8, 0.8, 0.8)

// is the difference between these two colors or data points greater than 0.1.
// i.e. is the difference large?
If (dist > 0.1) {
    // set the color to red
    gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```



The data in the above example isn't accurate and is emphasized for brevity and simplicity. But we check for the gradient or difference in data points and then set the color of the current fragment accordingly. Again it's also useful to point out that the

fragment code above would run for every square representation of a shader run or pixel.

Ok but why do we check for gradients or changes in color or data? Well edges are usually defined by changes in color due to the very nature of edges and gradients. Edges are just where something visually stops and another thing starts. Go look at an edge somewhere and you'll see what I mean. And this comparison of data or color is how we detect this change.

The third method is a novel approach to detect edges. It involves finding the edges of a 2D polygon and then colouring the polygon accordingly. We accomplish this by checking if a vertex is in front or behind

[*] Images and positional data**

[*] Procedural Flow maps**

Procedural Low-Poly Clouds

Real time vertex displacement

GLSL Floodfill and dynamic shaders

Pulsating volume

****River shader and tube geometries**

Spatial Clustering of Colors in Star Maps and Cell Images

Procedural Flow Maps

Appendix

1.1

```
import React, { useRef, useEffect } from 'react';
import { useThree, useFrame, extend } from '@react-three/fiber';
import { EffectComposer } from
'three/examples/jsm/postprocessing/EffectComposer';
import { ShaderPass } from
'three/examples/jsm/postprocessing/ShaderPass';
import { RenderPass } from
'three/examples/jsm/postprocessing/RenderPass';
import * as THREE from 'three';
```

```
extend({ EffectComposer, ShaderPass, RenderPass })
```

```
const shaderPass = {
  uniforms: {
    time: { value: 0 },
    tDiffuse: { value: null },
    depthTexture: { value: null },
    projectionMatrixInverse: { value: null },
    viewMatrixInverse: { value: null },
  },
  vertexShader: `
    varying vec2 vUv;
    void main () {
      vUv = uv;
      gl_Position = projectionMatrix * modelViewMatrix *
vec4(position, 1.0);
    }
  `,
  fragmentShader: `
    uniform float time;
    uniform sampler2D tDiffuse;
    uniform sampler2D depthTexture;
    varying vec2 vUv;

    uniform mat4 projectionMatrixInverse;
    uniform mat4 viewMatrixInverse;

    vec3 worldCoordinatesFromDepth(float depth) {
```

```

    float z = depth * 2.0 - 1.0;

    vec4 clipSpaceCoordinate = vec4(vUv * 2.0 - 1.0, z, 1.0);
    vec4 viewSpaceCoordinate = projectionMatrixInverse *
clipSpaceCoordinate;

    viewSpaceCoordinate /= viewSpaceCoordinate.w;

    vec4 worldSpaceCoordinates = viewMatrixInverse *
viewSpaceCoordinate;

    return worldSpaceCoordinates.xyz;
}

float sphereSDF(vec3 p, float radius) {
    return length(p) - radius;
}

void main() {
    float depth = texture( depthTexture, vUv ).x;
    vec3 worldPosition = worldCoordinatesFromDepth(depth);
    float radius = mod(0.1 * time * 10.0, 3.0);

    if (sphereSDF(worldPosition, radius) < 0.0 &&
sphereSDF(worldPosition, radius) > -1.0) {
        gl_FragColor = vec4(0.0,1.0,0.0,1.0);
    } else {
        vec3 sceneColor = texture(tDiffuse, vUv).xyz;
        gl_FragColor = vec4(sceneColor, 1.0);
    }
}

}

const Effects = () => {
    const composer = useRef();
    const ref = useRef()
    const { gl, size, scene, camera } = useThree();

    const [ target ] = React.useMemo(() => {
        const target = new THREE.WebGLRenderTarget(

```

```

        window.innerWidth,
        window.innerHeight,
        {
            minFilter: THREE.LinearFilter,
            magFilter: THREE.LinearFilter,
            format: THREE.RGBFormat,
            stencilBuffer: false,
            depthBuffer: true,
            depthTexture: new THREE.DepthTexture()
        },
    );
    return [ target ];
}, []);

useEffect(() => {
    composer.current.setSize(size.width, size.height)
}, [size])

useFrame((state) => {
    state.gl.setRenderTarget(target);
    state.gl.render(scene, camera);

    if (ref.current) {
        ref.current.uniforms['depthTexture'].value =target.depthTexture;
        ref.current.uniforms['projectionMatrixInverse'].value =
camera.projectionMatrixInverse;
        ref.current.uniforms['viewMatrixInverse'].value =
camera.matrixWorld;
        ref.current.uniforms['time'].value = state.clock.elapsedTime;
    }
    composer.current.render()
}, 1);

return (
    <effectComposer ref={composer} args={[gl]}>
        <renderPass attachArray="passes" scene={scene} camera={camera} />
        <shaderPass attachArray="passes" ref={ref} args={[shaderPass]}
needsSwap={false} renderToScreen />
    </effectComposer>
)

```

```
}
```

```
export default Effects;
```

1.2

```
import React, { useRef } from "react";
import * as THREE from "three";
import { OrbitControls } from
"three/examples/jsm/controls/OrbitControls";
import { extend, useThree, useFrame, useLoader } from
"@react-three/fiber";
import { TextureLoader } from "three/src/loaders/TextureLoader";
import img from "../green+grass-256x256.jpeg";
```

```
extend({ OrbitControls });
```

```
const vertexShader = `
  varying vec2 vUv;
  void main () {
    vUv = uv;
    gl_Position = projectionMatrix * modelViewMatrix * vec4(position,
1.0);
  }
`;
```

```
const fragmentShader = `
  uniform sampler2D tDiffuse;
  uniform vec2 resolution;
  varying vec2 vUv;

  float rand(vec2 co, float seed) {
    float a = 12.9898;
    float b = 78.233;
    float c = 43758.5453;
    float dt= dot(co.xy ,vec2(a,b));
    float sn= mod(dt, 3.14);
    return fract(sin(sn + seed) * c);
  }
`;
```

```

    vec2 rotateUV(vec2 uv, float rotation, vec2 mid) {
        return vec2(
            cos(rotation) * (uv.x - mid.x) + sin(rotation) * (uv.y - mid.y) +
mid.x,
            cos(rotation) * (uv.y - mid.y) - sin(rotation) * (uv.x - mid.x) +
mid.y
        );
    }

```

```

void main(void) {
    vec2 uv = vUv* 5.0;

    vec2 tile = floor(uv);

    vec2 center = tile + vec2(0.5, 0.5);

    vec2 randomRotatedTileUV = rotateUV(uv , rand(tile, 2.0)* 10.0,
center);

    gl_FragColor = texture2D(tDiffuse, randomRotatedTileUV);
}
;

```

```

const Map = () => {
    const {
        camera,
        size,
        gl: { domElement }
    } = useThree();

    const shaderRef = useRef(null);

    const [texture] = useLoader(THREE.TextureLoader, [img]);

    const uniforms = {
        tDiffuse: { value: null },
        resolution: { value: new THREE.Vector2(size.width, size.height) }
    };

```

```

console.log({ texture });

useFrame(() => {
  texture.wrapS = texture.wrapT = THREE.RepeatWrapping;

  if (shaderRef.current) {
    shaderRef.current.uniforms["tDiffuse"].value = texture;
  }
});
return (
  <>
    <orbitControls args={[camera, domElement]} />
    <mesh position={[0, 0, 0]} rotation=[[-Math.PI / 2, 0, 0]]>
      <planeBufferGeometry attach="geometry" args={[1000, 1000]} />
      <shaderMaterial
        attach="material"
        ref={shaderRef}
        args={[
          {
            uniforms,
            vertexShader,
            fragmentShader
          }
        ]}
      />
    </mesh>
  </>
);
};

export default Map;

```

1.3

```

import './styles.css';
import { memo, useCallback, useEffect, useRef } from "react";
import { Canvas, useFrame, useThree } from "@react-three/fiber";
import { OrbitControls } from "@react-three/drei";
import * as THREE from "three";
import { Scene } from "three";

```

```

// Loop cut at the mid points of triangles, either do onClick event on the mesh
// or do onPointMove and get the effect of painting verticies onto the mesh.

const Box = memo(() => {
  const geom = useRef();
  const mesh = useRef();
  const pointsGeom = useRef();
  const { scene } = useThree();

  // This is the main function, it deals with the buffer geometry and all of the complexities that
  // go with dealing with orders, verticies and triangle faces (order based)
  const getData = ({ e, geometry }) => {
    const newVertices = [];
    const newIndices = [];
    const numVertices = geometry.attributes.position.array.length / 3;

    const position = geometry.attributes.position;

    console.log({ e: e.intersections });
    // This is the face we have clicked or hovered on all coming through the useful
    // onclick event / onPointerMove event, which is actually a threejs raycaster with
    // optimizations
    const aIndex = e.intersections[0].face.a;
    const bIndex = e.intersections[0].face.b;
    const cIndex = e.intersections[0].face.c;

    // for the new verticies we are adding we dont want to reuse indicies as this would cause weird
    faces
    // for new verticies we can create new indicies and add them onto the end
    const newAIndex = numVertices;
    const newBIndex = numVertices + 1;
    const newCIndex = numVertices + 2;

    // This is grabbing the position using the intersections face indicies
    const aPosition = new THREE.Vector3().fromBufferAttribute(position, aIndex);
    const bPosition = new THREE.Vector3().fromBufferAttribute(position, bIndex);
    const cPosition = new THREE.Vector3().fromBufferAttribute(position, cIndex);

    // we can then find the mid point between each corner
    const abPosition = new THREE.Vector3().lerpVectors(
      aPosition,
      bPosition,
      0.5
    );
    const bcPosition = new THREE.Vector3().lerpVectors(
      bPosition,
      cPosition,

```

```

    0.5
  );
  const caPosition = new THREE.Vector3().lerpVectors(
    cPosition,
    aPosition,
    0.5
  );

  // and finally we add the indices and vertices to the arrays

  // if we add midpoints between a triangle we go from 1 triangle to 4 which requires
  // 3 vertices for each triangle and therefore 12 in total indices are added.
  // IMPORTANT: they come in 3 and the order really does matter 1 for each corner of the triangle
  // and it is completely ok for one part of this triplet group of indices to be reused in other groups.
  // hence interleaved

  // Just remember 3 indexes = 1 face drawn
  newIndices.push(
    aIndex,
    newAIndex,
    newCIndex,
    newAIndex,
    bIndex,
    newBIndex,
    newAIndex,
    newBIndex,
    newCIndex,
    newBIndex,
    cIndex,
    newCIndex
  );

  // we only want to add new vertex's not the old ones otherwise we get duplication
  newVertices.push(
    abPosition.x,
    abPosition.y,
    abPosition.z,
    bcPosition.x,
    bcPosition.y,
    bcPosition.z,
    caPosition.x,
    caPosition.y,
    caPosition.z
  );

  return { newVertices, newIndices };
};

```

```

function filterArray(source, arr) {
  const result = [];
  for (let i = 0; i < arr.length; i += 3) {
    const index1 = arr[i];
    const index2 = arr[i + 1];
    const index3 = arr[i + 2];

    if (
      source.includes(index1) &&
      source.includes(index2) &&
      source.includes(index3)
    ) {
      continue;
    }

    result.push(index1, index2, index3);
  }
  return result;
}

const loopCut = ({ geometry, e }) => {
  const { newVertices, newIndices } = getData({ e, geometry });

  // we have to reduce the overall indices count as when we generate them above,
  // we have to remove the old face and add the new faces which make up the old face ]
  geometry.index.count -= 3;

  const positions = geometry.attributes.position.array;

  let indices = geometry.index.array;

  // As we are updating the number of vertices I found you have to set the draw range to the
  // new total number of vertices which is roughly less than the total of old + new indices
  geometry.setDrawRange(0, indices.length + newIndices.length);

  // setting the position attribute with the updated positions
  geometry.setAttribute(
    "position",
    new THREE.Float32BufferAttribute(
      new Float32Array([...positions, ...newVertices]),
      3
    )
  );

  // As stated we have to remove the old face otherwise it comes back in the onClick /
  onPointerMove events

```

// and it tries to subdivide the same face which results in no extra [points but more verticies in the same position

```
const filteredArrIndex = filterArray(  
  [  
    e.intersections[0].face.a,  
    e.intersections[0].face.b,  
    e.intersections[0].face.c,  
  ],  
  indices  
);
```

```
// Setting the index's which determine face rendering  
geometry.setIndex(  
  new THREE.BufferAttribute(  
    new Uint32Array([...filteredArrIndex, ...newIndices]),  
    1  
  )  
);
```

// Probs dont need all this and abit gungho but ah well

// nobody knows everything haha

```
geometry.verticesNeedUpdate = true;  
geometry.attributes.position.needsUpdate = true;  
geometry.index.needsUpdate = true;  
geometry.computeFaceNormals = true;  
geometry.computeVertexNormals();  
geometry.computeBoundingSphere();  
geometry.computeBoundingBox();  
};
```

```
return (  
  <group>  
    <mesh  
      ref={mesh}  
      // Change to onClick in you want to  
      onPointerMove={(e) => {  
        // One to visualise the verticies, Points  
        // and the Cube for visually looking at the geometry  
        loopCut({ e, geometry: geom.current, meshA: mesh.current, scene });  
        loopCut({  
          e,  
          geometry: pointsGeom.current,  
          meshA: mesh.current,  
          scene,  
        });  
      }}  
    </mesh>  
  </group>  
  scale=[2, 2, 2])
```

```

        needsUpdate
    >
    <boxBufferGeometry ref={geom} />
    <meshBasicMaterial needsUpdate />
</mesh>
<points scale={[2, 2, 2]}>
    <boxBufferGeometry ref={pointsGeom} />
    <pointsMaterial size={0.04} color={"red"} />
</points>
</group>
);
});

```

1.4

```

export const fragmentShader = `
uniform float time;
uniform sampler2D grassTexture;

varying vec3 pos;
varying vec2 vUv;

${noise3D}

void main () {
    float scaledNoise = .0190;
    float noise = noise(pos);

    float wavyStrength = sin(time + pos.y)* 0.04;
    vec2 uvs = vec2(mix(vUv.x, noise * scaledNoise , wavyStrength ), vUv.y);

    float alpha = texture2D(grassTexture, uvs).r;

    //If transparent, don't draw
    if (alpha < 0.005) discard;

    vec4 color = texture2D(grassTexture, uvs.xy);
    gl_FragColor = color;
}
`;

export const vertexShader = `
varying vec2 vUv;

```

```

varying vec3 pos;

void main () {
  vUv = uv;
  pos = position;
  gl_Position = projectionMatrix * modelViewMatrix * vec4(position, 1.0);
}
`;

const GrassMaterial = shaderMaterial(
  {
    time: 0,
    grassTexture: null
  },
  vertexShader,
  fragmentShader
  // (material) => {
  //   material.transparent = true; // enable transparency
  //   material.alphaTest = 0.5; // set alpha test value (optional)
  //   material.depthWrite = false; // disable writing to depth buffer (optional)
  // }
);

extend({ GrassMaterial });

const Grass = () => {
  const meshRef = useRef(null);
  const { clock } = useThree();

  const [grassTexture] = useLoader(THREE.TextureLoader, ["/grass_4.png"]);

  useFrame(() => {
    if (meshRef.current.material) {
      meshRef.current.material.uniforms.time.value = clock.getElapsedTime();
      meshRef.current.material.uniforms.grassTexture.value = grassTexture;
    }
  });

  return (
    <mesh ref={meshRef}>
      <planeGeometry args={[10, 10, 10]} />
      <grassMaterial transparent />
    </mesh>
  );
}

```

```
);  
};
```

```
export default Grass;
```

1.5

```
uniform sampler2D tDiffuse;  
uniform sampler2D depthTexture;  
uniform vec3 camPos;  
uniform mat4 cameraProjectionMatrixInverse;  
uniform mat4 cameraWorldMatrix;  
varying vec3 pos;  
varying vec2 vUv;  
  
const int NUM_SAMPLES = 150;  
vec3 light_on_screen = vec3(0.5, .5, 0.0);  
  
vec3 worldCoordinatesFromDepth(float depth) {  
    float z = depth * 2.0 - 1.0;  
  
    vec4 clipSpaceCoordinate = vec4(vUv * 2.0 - 1.0, z, 1.0);  
    vec4 viewSpaceCoordinate = cameraProjectionMatrixInverse * clipSpaceCoordinate;  
  
    viewSpaceCoordinate /= viewSpaceCoordinate.w;  
  
    vec4 worldSpaceCoordinates = cameraWorldMatrix * viewSpaceCoordinate;  
  
    return worldSpaceCoordinates.xyz;  
}  
  
void main() {  
    const float exposure = 0.055;  
    const float decay = .99;  
    const float density = 0.54;  
    const float weight = 4.75;  
  
    // Inner used valuesa
```

```

vec2 deltaTextCoord = vec2(vUv.xy - light_on_screen.xy);
vec2 textCoo = vUv.st;
deltaTextCoord *= 1.0 / float(NUM_SAMPLES) * density;
float illuminationDecay = .10;

vec4 c = vec4(0.0, 0.0, 0.0, 1.0);

for(int i=0; i < NUM_SAMPLES ; i++) {
    textCoo -= deltaTextCoord;

    textCoo.s = clamp(textCoo.s, 0.0, 1.0);
    textCoo.t = clamp(textCoo.t, 0.0, 1.0);

    vec4 sample1 = texture2D(tDiffuse, textCoo);

    sample1 *= illuminationDecay * weight;

    c += sample1;

    illuminationDecay *= decay;
}

c *= exposure;

c.r = clamp(c.r, 0.0, 1.0);
c.g = clamp(c.r, 0.0, 1.0);
c.b = clamp(c.r, 0.0, 1.0);

vec4 textureSample = texture2D(tDiffuse, vUv);
float depth = texture2D(depthTexture, vUv).r;
vec3 worldPos = worldCoordinatesFromDepth(depth);

if (length(worldPos) > length(camPos) || camPos.z >= worldPos.z) {
    gl_FragColor = vec4(c.rgba);
} else {
    gl_FragColor = textureSample.rgba;
}
}

```

1.5/1.6

```
//  
https://codesandbox.io/s/misty-hooks-yury3?from-embed=&file=/src/Effects/Shader/Shader.  
js:895-896  
// https://github.com/ycw/three-radial-blur/blob/main/src/RadialBlurShader.js  
  
import React, { useMemo, useRef } from "react";  
import { useThree, useFrame } from "@react-three/fiber";  
import { EffectComposer } from "three/examples/jsm/postprocessing/EffectComposer";  
import { RenderPass } from "three/examples/jsm/postprocessing/RenderPass";  
import { ShaderPass } from "three/examples/jsm/postprocessing/ShaderPass";  
import CombiningShader from "../shaders/CombiningShader";  
import * as THREE from "three";  
import CarModel from "../model/house";  
  
export const RadialBlurShader = {  
  defines: {  
    MAX_ITERATIONS: 100  
  },  
  uniforms: {  
    tDiffuse: { value: null },  
    depthTexture: { value: null },  
    radialCenter: { value: null },  
    intensity: { value: 10.0 },  
    iterations: { value: 1 },  
    uResolution: { value: null },  
    camPos: { value: null },  
    cameraWorldMatrix: { value: null },  
    cameraProjectionMatrixInverse: { value: null },  
    windowPos: { value: null }  
  },  
  vertexShader: /* glsl */ `  
    uniform vec3 windowPos;  
    varying vec2 vUv;  
    varying vec3 pos;  
    void main() {  
      vUv = uv;
```

```

    pos = (projectionMatrix * modelViewMatrix * vec4( position, 1.0 )).xyz;
    gl_Position = projectionMatrix * modelViewMatrix * vec4( position, 1.0 );
}
,
fragmentShader: /* glsl */ `
    uniform sampler2D tDiffuse;
    uniform sampler2D depthTexture;
    uniform vec3 camPos;
    uniform mat4 cameraProjectionMatrixInverse;
    uniform mat4 cameraWorldMatrix;
    varying vec3 pos;
    varying vec2 vUv;

    const int NUM_SAMPLES = 150;
    vec3 light_on_screen = vec3(0.5, .5, 0.0);

    vec3 worldCoordinatesFromDepth(float depth) {
        float z = depth * 2.0 - 1.0;

        vec4 clipSpaceCoordinate = vec4(vUv * 2.0 - 1.0, z, 1.0);
        vec4 viewSpaceCoordinate = cameraProjectionMatrixInverse * clipSpaceCoordinate;

        viewSpaceCoordinate /= viewSpaceCoordinate.w;

        vec4 worldSpaceCoordinates = cameraWorldMatrix * viewSpaceCoordinate;

        return worldSpaceCoordinates.xyz;
    }

    void main() {
        const float exposure = 0.055;
        const float decay = .99;
        const float density = 0.54;
        const float weight = 4.75;

        // Inner used valuesa
        vec2 deltaTextCoord = vec2(vUv.xy - light_on_screen.xy);
        vec2 textCoo = vUv.st;
        deltaTextCoord *= 1.0 / float(NUM_SAMPLES) * density;
        float illuminationDecay = .10;

        vec4 c = vec4(0.0, 0.0, 0.0, 1.0);

```

```

for(int i=0; i < NUM_SAMPLES ; i++)
{
    textCoo -= deltaTextCoord;

    textCoo.s = clamp(textCoo.s, 0.0, 1.0);
    textCoo.t = clamp(textCoo.t, 0.0, 1.0);

    vec4 sample1 = texture2D(tDiffuse, textCoo);

    sample1 *= illuminationDecay * weight;

    c += sample1;

    illuminationDecay *= decay;
}

c *= exposure;

c.r = clamp(c.r, 0.0, 1.0);
c.g = clamp(c.g, 0.0, 1.0);
c.b = clamp(c.b, 0.0, 1.0);

vec4 textureSample = texture2D(tDiffuse, vUv);
float depth = texture2D(depthTexture, vUv).r;
vec3 worldPos = worldCoordinatesFromDepth(depth);

if (length(worldPos) > length(camPos) || camPos.z >= worldPos.z) {
    gl_FragColor = vec4(c.rgba);
} else {
    gl_FragColor = textureSample.rgba;
}

}

,
};

const Effects = ({ children }) => {
    const { gl, scene, camera, size } = useThree();

    const materialRef = useRef(null);

    const [target] = React.useMemo(() => {

```

```

const target = new THREE.WebGLRenderTarget(
  window.innerWidth,
  window.innerHeight,
  {
    minFilter: THREE.LinearFilter,
    magFilter: THREE.LinearFilter,
    format: THREE.RGBFormat,
    stencilBuffer: false,
    depthBuffer: true,
    depthTexture: new THREE.DepthTexture()
  }
);
return [target];
}, []);

const [radialBlurComposer, finalComposer, radialBlur] = useMemo(() => {
  const renderScene = new RenderPass(scene, camera);

  const occlusionRenderTarget = new THREE.WebGLRenderTarget(
    window.innerWidth * 0.5,
    window.innerHeight * 0.5
  );

  const radialBlurComposer = new EffectComposer(gl, occlusionRenderTarget);
  radialBlurComposer.renderToScreen = false;
  radialBlurComposer.addPass(renderScene);

  const radialBlur = new ShaderPass(RadialBlurShader);
  radialBlur.needsSwap = false;
  radialBlur.uniforms.radialCenter.value = new THREE.Vector2(0.5, 0.5);
  radialBlur.uniforms.intensity.value = 10;
  radialBlur.uniforms.iterations.value = 300;
  radialBlurComposer.addPass(radialBlur);

  const finalComposer = new EffectComposer(gl);
  finalComposer.addPass(new RenderPass(scene, camera));

  const additivePass = new ShaderPass(CombiningShader());
  additivePass.uniforms.tAdd.value = occlusionRenderTarget.texture;
  additivePass.needsSwap = false;
  finalComposer.addPass(additivePass);

  return [radialBlurComposer, finalComposer, radialBlur, additivePass];

```

```

}, []);

useFrame((state) => {
  state.gl.setRenderTarget(target);
  state.gl.render(scene, camera);
  camera.layers.set(2);
  radialBlur.uniforms["depthTexture"].value = target.depthTexture;
  radialBlur.uniforms["uResolution"].value = new THREE.Vector2(
    size.width,
    size.height
  );
  radialBlur.uniforms["camPos"].value = new THREE.Vector3().copy(
    camera.position
  );
  radialBlur.uniforms["cameraWorldMatrix"].value = camera.matrixWorld;
  radialBlur.uniforms[
    "cameraProjectionMatrixInverse"
  ].value = new THREE.Matrix4().copy(camera.projectionMatrix).invert();

  radialBlur.uniforms["windowPos"].value = new THREE.Vector3(-2, 0, 0);
  radialBlurComposer.render();
  camera.layers.set(1);
  finalComposer.render();
}, 1);

return <CarModel materialRef={materialRef} />;
};

export default Effects;

```